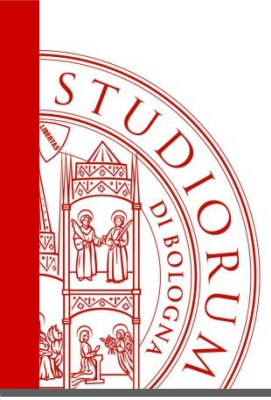


Università degli Studi di Bologna
Corso di Laurea in Ingegneria Informatica

Principi di Design

Ingegneria del Software T

Prof. MARCO PATELLA
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Qualità della progettazione

- La **Qualità della progettazione** è un concetto vago
- La qualità dipende da specifiche **priorità dell'organizzazione**
- Un 'buon' progetto potrebbe essere
  - il più affidabile,
  - il più efficiente,
  - il più manutenibile,
  - il più economico, …
- Gli argomenti qui discussi riguardano principalmente la **manutenibilità del progetto**

Cosa rende un design “cattivo”?

- ✓ **Misdirection**: non soddisfa i requisiti
- ✓ **Rigidità del software**: una singola modifica influisce su molte altre parti del sistema
- ✓ **Fragilità del software**: una singola modifica influisce su parti inaspettate del sistema
- ✓ **Immobilità del software**: è difficile da riutilizzare in un’altra applicazione
- ✓ **Viscosità**: è difficile fare la cosa giusta, ma facile fare la cosa sbagliata

Rigidità del Software

- La tendenza per il software a essere **difficile da modificare**
- **Sintomo**: ogni modifica provoca una cascata di modifiche successive nei moduli dipendenti
- **Effetto**: i manager hanno timore ad accettare che gli sviluppatori risolvano problemi non critici – non sanno se/quando gli sviluppatori termineranno le modifiche

Fragilità del Software

- La tendenza del software a **“rompersi” in molti punti** ogni volta che viene modificato ▶ i cambiamenti tendono a causare **comportamenti inaspettati** in altre parti del sistema (spesso in aree che non hanno alcuna relazione concettuale con l’area che è stata modificata)
- **Sintomo**: ogni correzione peggiora le cose, introducendo più problemi di quelli risolti ▶ tale software è impossibile da manutenere
- **Effetto**: ogni volta che i manager autorizzano una correzione, temono che il software si “rompa” in modo inaspettato

Immobilità del Software

- L'**impossibilità di riutilizzare il software** di altri progetti o di parti dello stesso progetto
- **Sintomo**: uno sviluppatore scopre di aver bisogno di un modulo simile a quello scritto da un altro sviluppatore. Ma il modulo in questione ha troppe dipendenze. Dopo molto lavoro, lo sviluppatore scopre che il lavoro e il rischio necessari per separare le parti desiderabili del software dalle parti indesiderabili sono troppo grandi per essere tollerati
- **Effetto**: e così il software viene semplicemente riscritto anziché riutilizzato

Viscosità del Software

- Gli sviluppatori di solito trovano più di un modo per apportare una modifica
  - alcuni preservano il design, altri no (cioè sono hack)
- La tendenza a **incoraggiare modifiche al software che sono hack** piuttosto che modifiche al software che preservano l'intento di progettazione originale
  - **Viscosità del design**: i metodi che preservano il design sono più difficili da utilizzare rispetto agli hack
  - **Viscosità dell'ambiente**: l'ambiente di sviluppo è lento e inefficiente (tempi di compilazione molto lunghi, il sistema di controllo dei sorgenti richiede ore per archiviare pochi file, ...)
- **Sintomo**: è facile fare la cosa sbagliata, ma difficile fare la cosa giusta
- **Effetto**: la manutenibilità del software degenera a causa di hack, scorciatoie, correzioni temporanee, ...

Perché esistono risultati di progettazione scadenti?

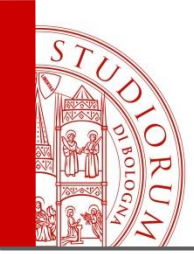
- Ragioni ovvie:
  - mancanza di capacità/pratica di progettazione
  - tecnologie in evoluzione
  - vincoli di tempo/risorse
  - complessità del dominio, ...
- Non così ovvie:
  - **la "putrefazione" del software è un processo lento** – anche un design originariamente pulito ed elegante può degenerare nel corso dei mesi/anni
  - i **requisiti** spesso **cambiano** in modi che non erano stati previsti dal design (o dal progettista) originale
  - le **dipendenze** tra moduli non pianificate e improprie si insinuano ▶ le dipendenze non vengono gestite

Modifiche ai Requisiti

- Come ingegneri del software, sappiamo benissimo che i requisiti cambiano
- In effetti, la maggior parte di noi si rende conto che il documento dei requisiti è il **documento più volatile** dell'intero progetto
- Se i nostri progetti falliscono a causa del costante arrivo di requisiti in continua evoluzione, **la colpa è della nostra progettazione**
- Dobbiamo in qualche modo trovare un modo per rendere i nostri progetti resistenti a tali cambiamenti e proteggerli dalla putrefazione

Gestione delle Dipendenze

- Ciascuno dei quattro sintomi sopra menzionati è causato (direttamente o indirettamente) da **dipendenze improprie tra moduli** del software
- È l'**architettura delle dipendenze** che si sta degradando e con essa la capacità del software di essere manutenuto
- Per prevenire il degrado dell'architettura delle dipendenze, è necessario gestire le dipendenze tra i moduli in un'applicazione
- La progettazione orientata agli oggetti è piena di **principi** e **tecniche** per la gestione delle dipendenze dei moduli



Principi di Design

- The **Single Responsibility Principle** (SRP)
- The **Dependency Inversion Principle** (DIP)
- The **Interface Segregation Principle** (ISP)
- The **Open/Closed Principle** (OCP)
- The **Liskov Substitution Principle** (LSP)

Premessa
Il principio zero

- Il principio zero è un principio di logica noto come **rasoio di Occam**:
  "*Entia non sunt multiplicanda praeter necessitatem*"
  ovvero: **non bisogna introdurre concetti che non siano strettamente necessari**
- È la forma "colta" di un principio pratico:
  "**Quello che non c'è non si rompe**" (H. Ford)
- Tra due soluzioni bisogna preferire quella
  - che introduce il minor numero di ipotesi e
  - che fa uso del minor numero di concetti

Premessa
Semplicità e semplicismo

- La **semplicità** è un fattore importantissimo
  - il software deve fare i conti con una notevole componente di complessità, generata dal contesto in cui deve essere utilizzato
  - quindi è estremamente importante **non aggiungere altra complessità** arbitraria
- Il problema è che
  - la semplicità richiede uno **sforzo non indifferente** (**è molto più facile essere complicati che semplici**)
  - in generale le soluzioni più semplici vengono in mente per ultime
- Bisogna fare poi molta attenzione a essere semplici ma non semplicistici
  "**Keep it as simple as possible but not simpler**" (A. Einstein)

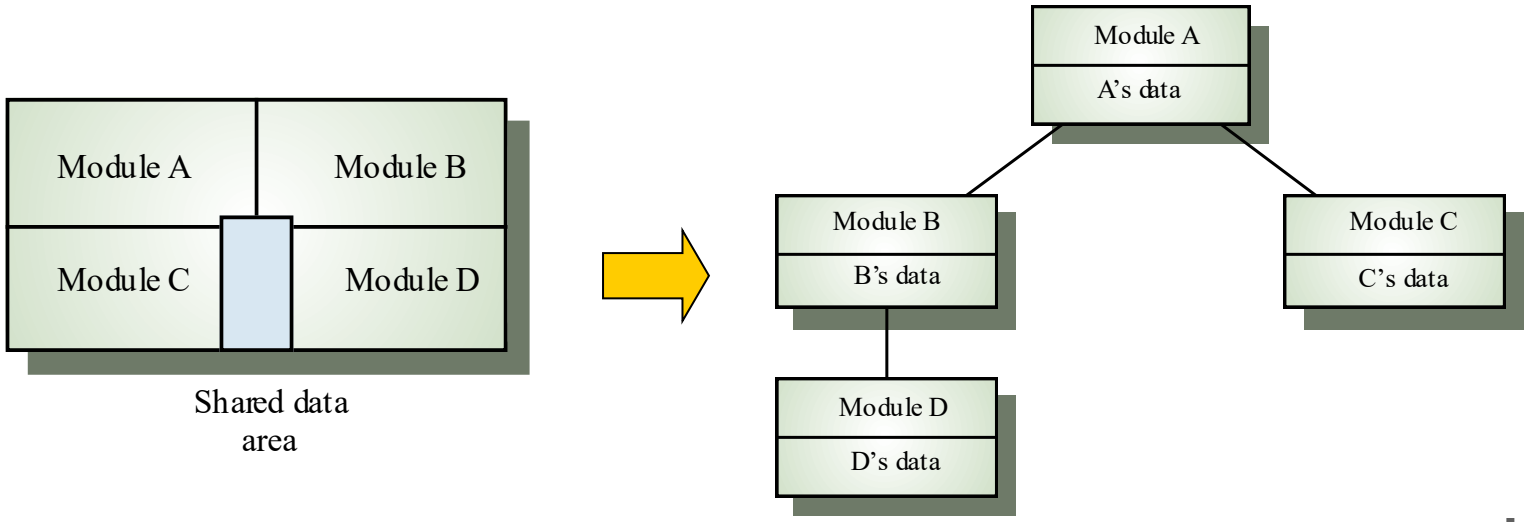
Premessa
Divide et impera

- La **decomposizione** è una tecnica fondamentale per il controllo e la gestione della complessità
- Non esiste un solo modo per decomporre il sistema
  ▶ la **qualità della progettazione** dipende direttamente dalla **qualità delle scelte di decomposizione** adottate
- In questo contesto il **principio fondamentale** è:
  minimizzare il grado di accoppiamento tra i moduli del sistema
- Da questo principio è possibile ricavare diverse regole:
  - minimizzare la quantità di interazione fra i moduli
  - eliminare tutti i riferimenti circolari fra moduli
  - …

Premessa

Rendere privati tutti i dati degli oggetti

- Le modifiche ai dati pubblici rischiano sempre di “aprire” il modulo:
  - possono avere un effetto domino che porta a richiedere modifiche in molte posizioni impreviste
  - gli errori possono essere difficili da trovare e correggere completamente – le correzioni possono provocare errori altrove

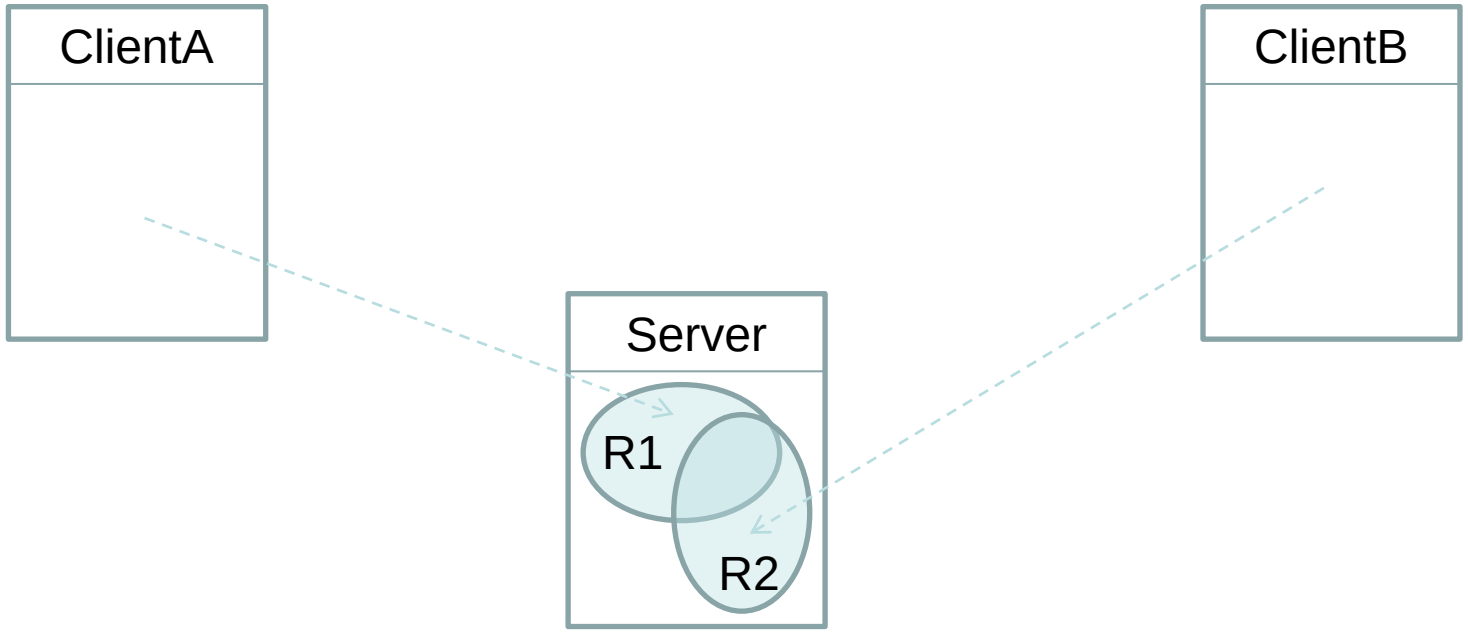


The Single Responsibility Principle

- ***There should never be more than one reason for a class to change*** (R. Martin)
- ***A class has a single responsibility: it does it all, does it well, and does it only*** (1-Responsibility Rule)

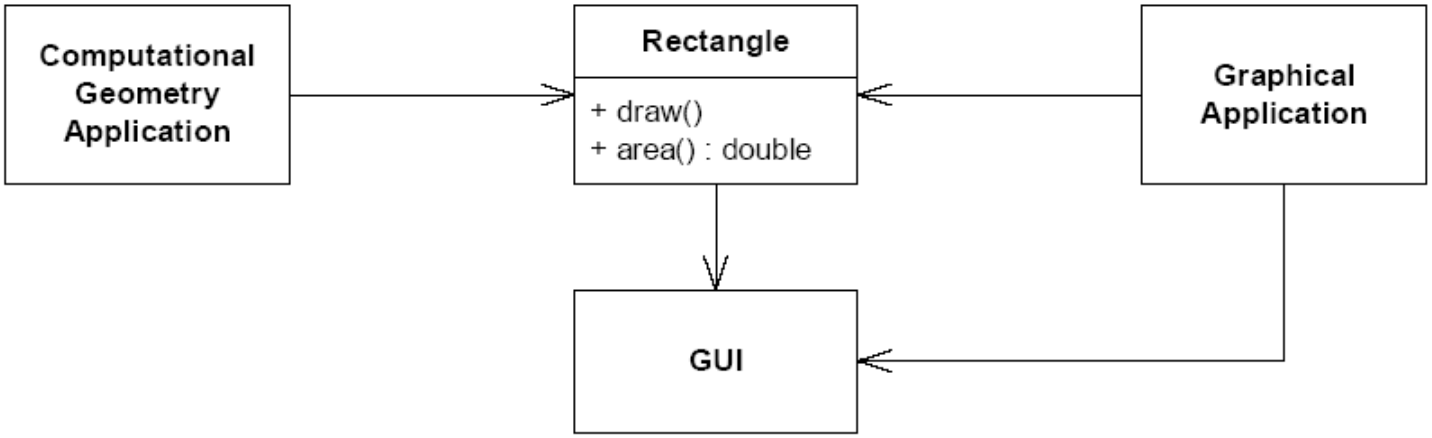
- Se una classe ha più di una responsabilità, queste diventano accoppiate
- Le modifiche a una responsabilità possono compromettere o inibire la capacità della classe di realizzare le altre
- Questo tipo di accoppiamento porta a **design fragili** che si rompono in modi inaspettati quando modificati

The Single Responsibility Principle



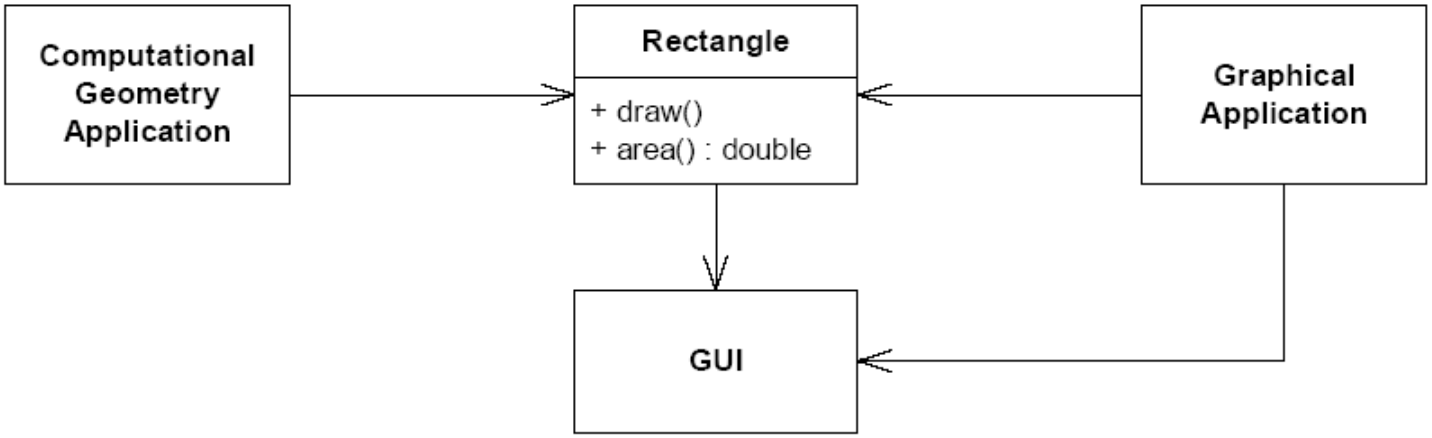
Una modifica di R1 può avere delle conseguenze anche su ClientB che NON utilizza direttamente R1

The Single Responsibility Principle
Esempio



- Un'applicazione è di geometria computazionale
  - usa `Rectangle` per aiutarsi con la matematica delle forme geometriche
  - non disegna mai il rettangolo sullo schermo
- L'altra applicazione è di natura grafica
  - può anche fare un po' di geometria computazionale, ma
  - disegna sicuramente il rettangolo sullo schermo

The Single Responsibility Principle
Esempio

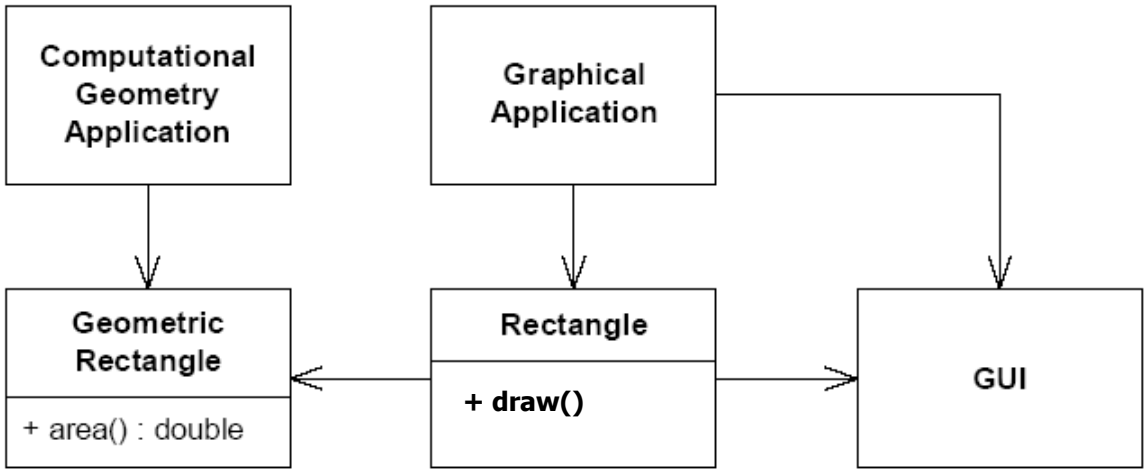


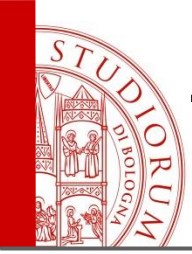
- La classe **`Rectangle`** ha **due responsabilità**
  - la **prima responsabilità** è fornire un modello matematico della geometria di un rettangolo
  - la **seconda responsabilità** la seconda responsabilità è disegnare il rettangolo su un'interfaccia grafica

The Single Responsibility Principle
Esempio – **Refactoring**

- Un progetto migliore consiste nel **separare le due responsabilità in due classi completamente distinte**

- **Si estrae una classe**: si crea una nuova classe e si spostano i campi e i metodi opportuni dalla vecchia classe alla nuova classe



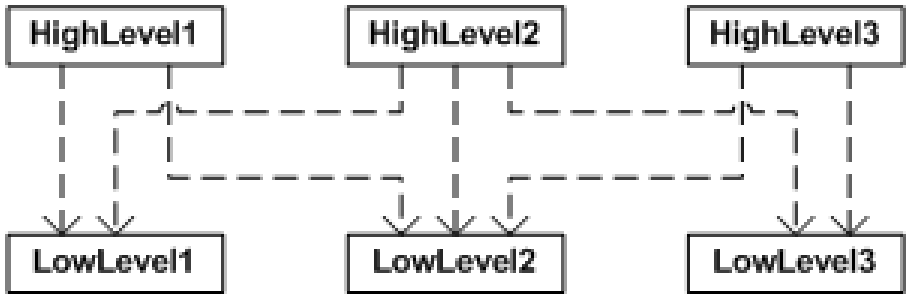


The Dependency Inversion Principle

- ***Depend upon abstractions***
  Do not depend upon concretions

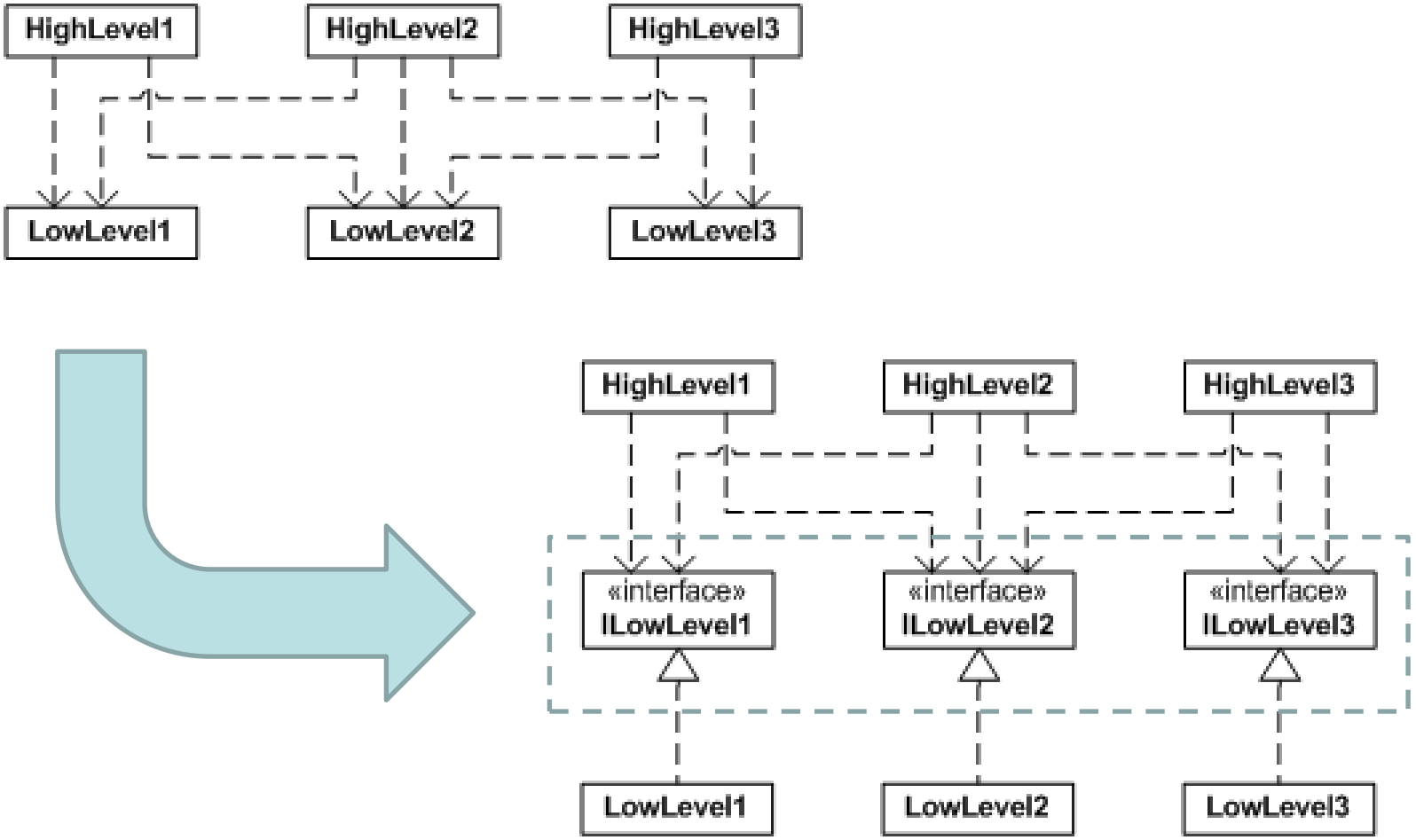
- Ogni dipendenza dovrebbe puntare a un'interfaccia o a una classe astratta
- Nessuna dipendenza dovrebbe puntare a una classe concreta
- I moduli di alto livello (i clienti) non dovrebbero dipendere dai moduli di basso livello (i fornitori di servizi)
  Entrambi dovrebbero dipendere da astrazioni

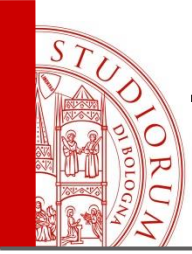
The Dependency Inversion Principle



- I **moduli di basso livello** contengono la maggior parte del codice e della logica implementativa e quindi **sono i più soggetti a cambiamenti**
- Se **i moduli di alto livello** dipendono dai dettagli dei moduli di basso livello (sono accoppiati in modo troppo stretto), i cambiamenti si propagano e le conseguenze sono:
  - **Rigidità**: bisogna intervenire su un numero elevato di moduli
  - **Fragilità**: si introducono errori in altre parti del sistema
  - **Immobilità**: i moduli di alto livello non si possono riutilizzare perché non si riescono a separare da quelli di basso livello

The Dependency Inversion Principle





The Dependency Inversion Principle

- Questo principio funziona perché:
  - **le astrazioni** contengono pochissimo codice (in teoria nulla) e quindi **sono poco soggette a cambiamenti**
  - i **moduli non astratti sono soggetti a cambiamenti** ma questi cambiamenti sono sicuri perché nessuno dipende da questi moduli
- I dettagli del sistema sono stati isolati, separati da un **muro di astrazioni stabili**, e questo impedisce ai cambiamenti di propagarsi (**design for change**)
- Nel contempo i singoli moduli sono **maggiormente riusabili** perché sono disaccoppiati fra di loro (**design for reuse**)

The Dependency Inversion Principle

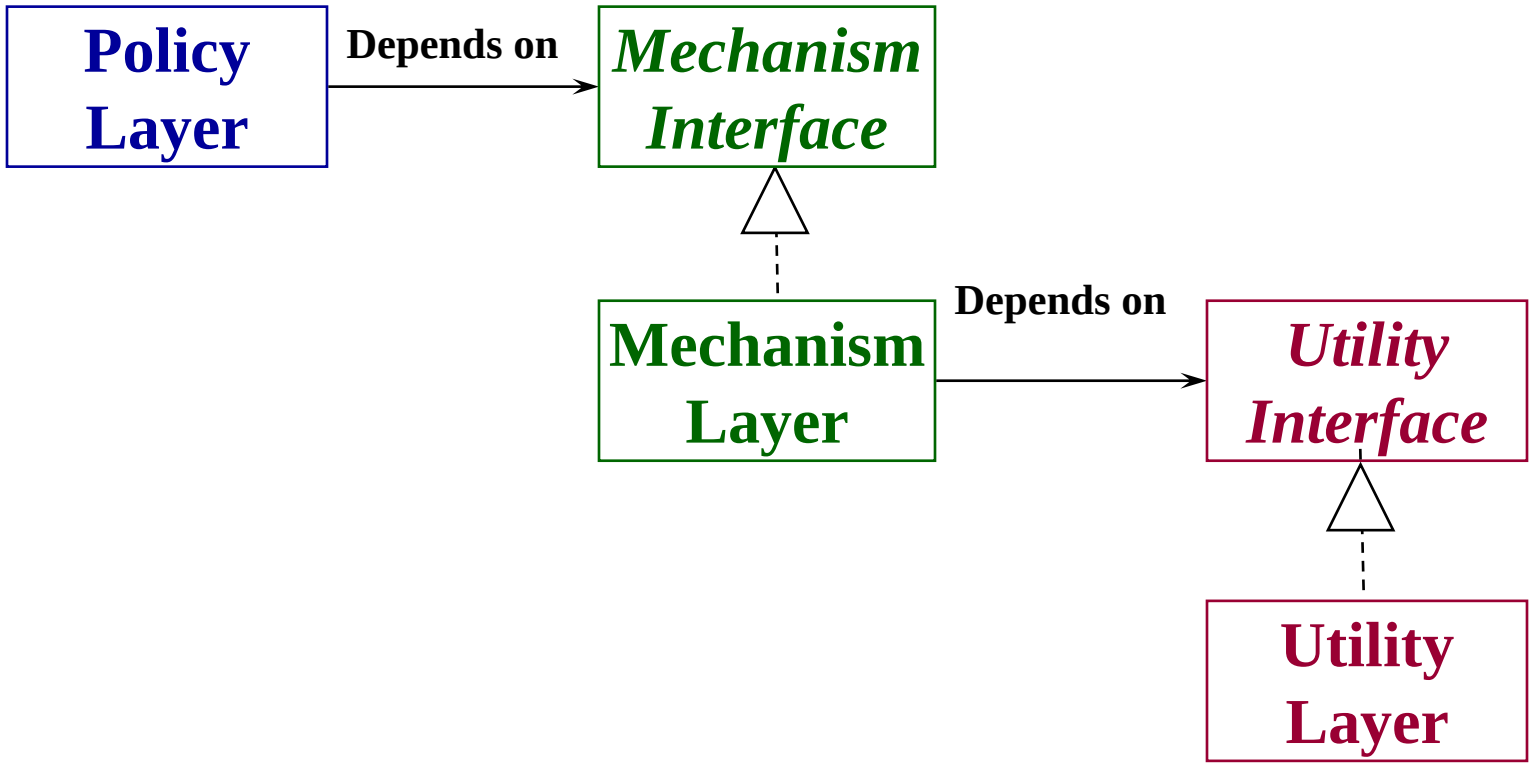
Dipendenze transitive

- "***...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface***" (Grady Booch)

- I sistemi software dovrebbero essere stratificati, cioè organizzati a livelli
- Le **dipendenze transitive** devono essere eliminate

The Dependency Inversion Principle

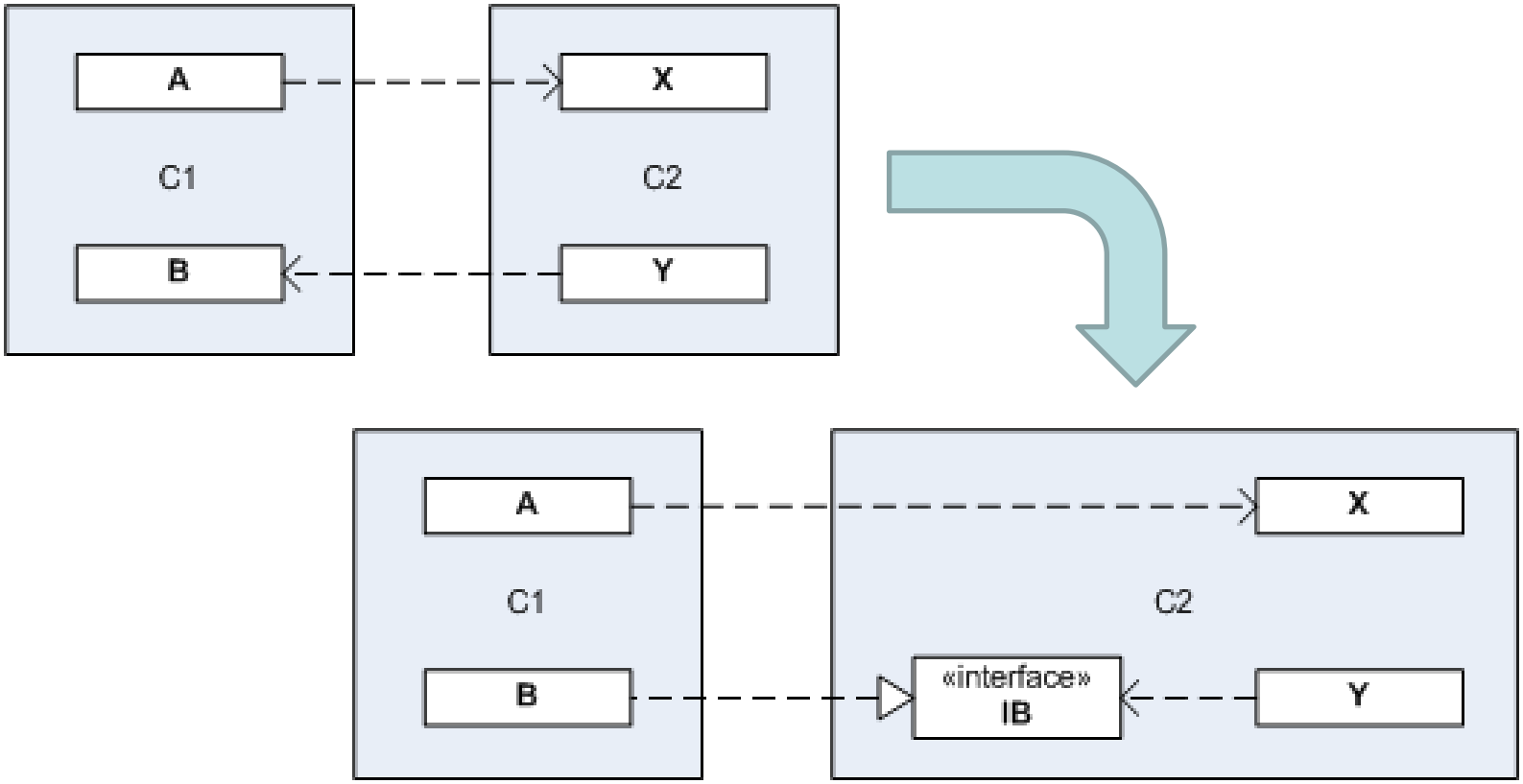
Dipendenze transitive



The Dependency Inversion Principle

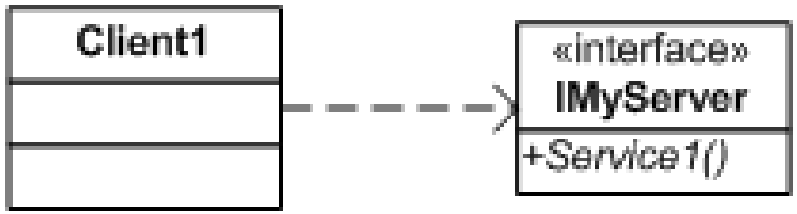
Dipendenze cicliche

- Le **dipendenze cicliche** devono essere eliminate

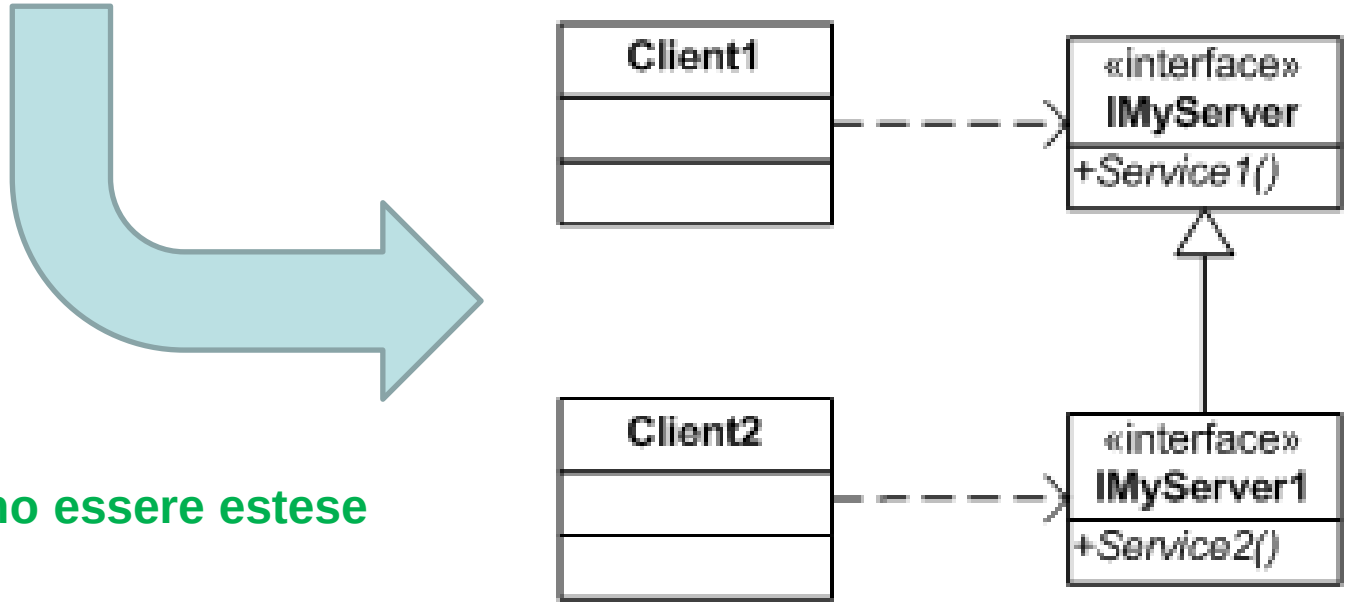


The Dependency Inversion Principle

Stabilità delle astrazioni



Le astrazioni in uso **NON devono essere modificate**



Ma **devono essere estese**

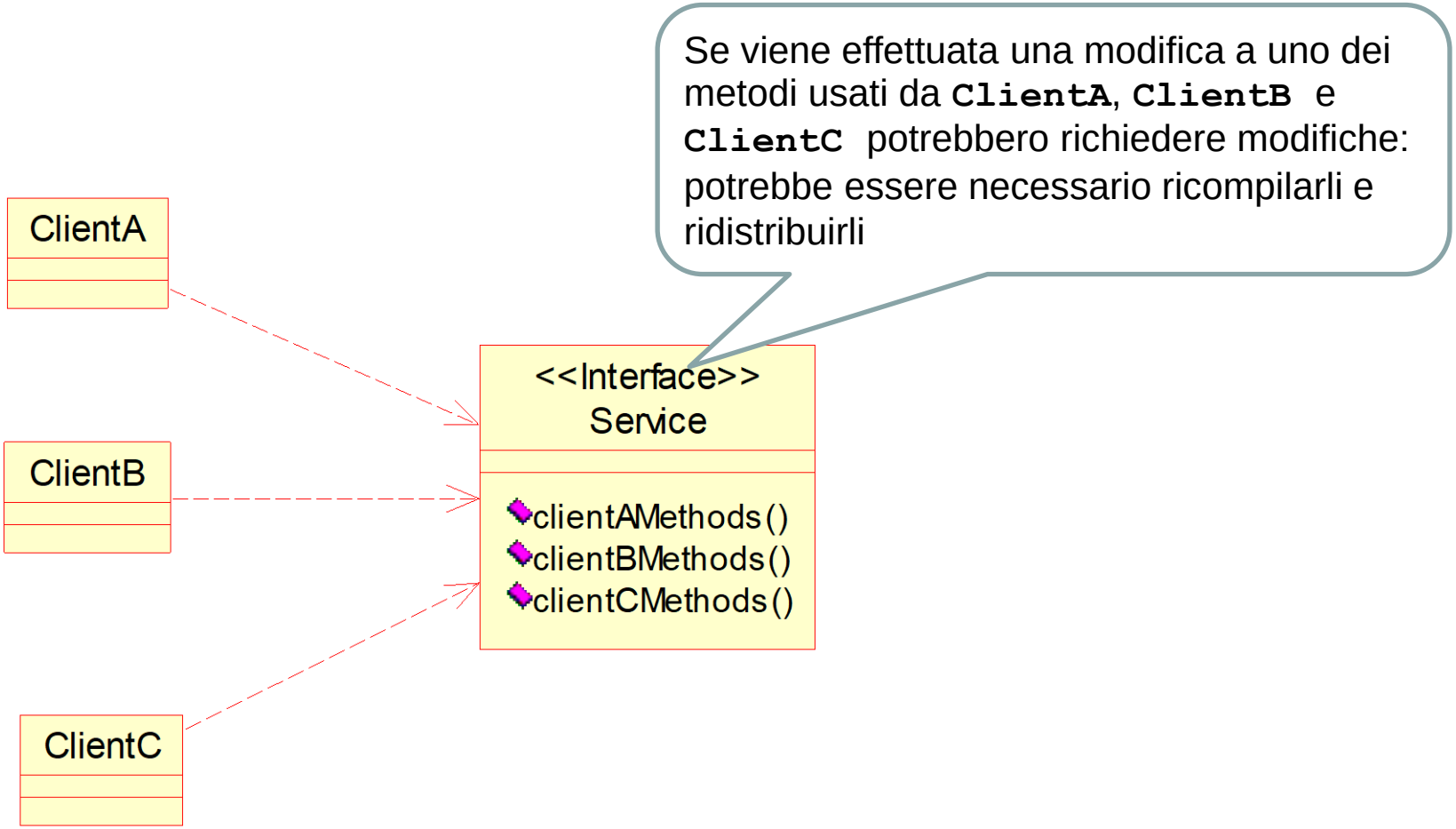
The Interface Segregation Principle

- ***Clients should not be forced to depend upon interfaces that they do not use***

- Molte interfacce specifiche per un cliente sono meglio di un’unica interfaccia general purpose

The Interface Segregation Principle

Fat Interface



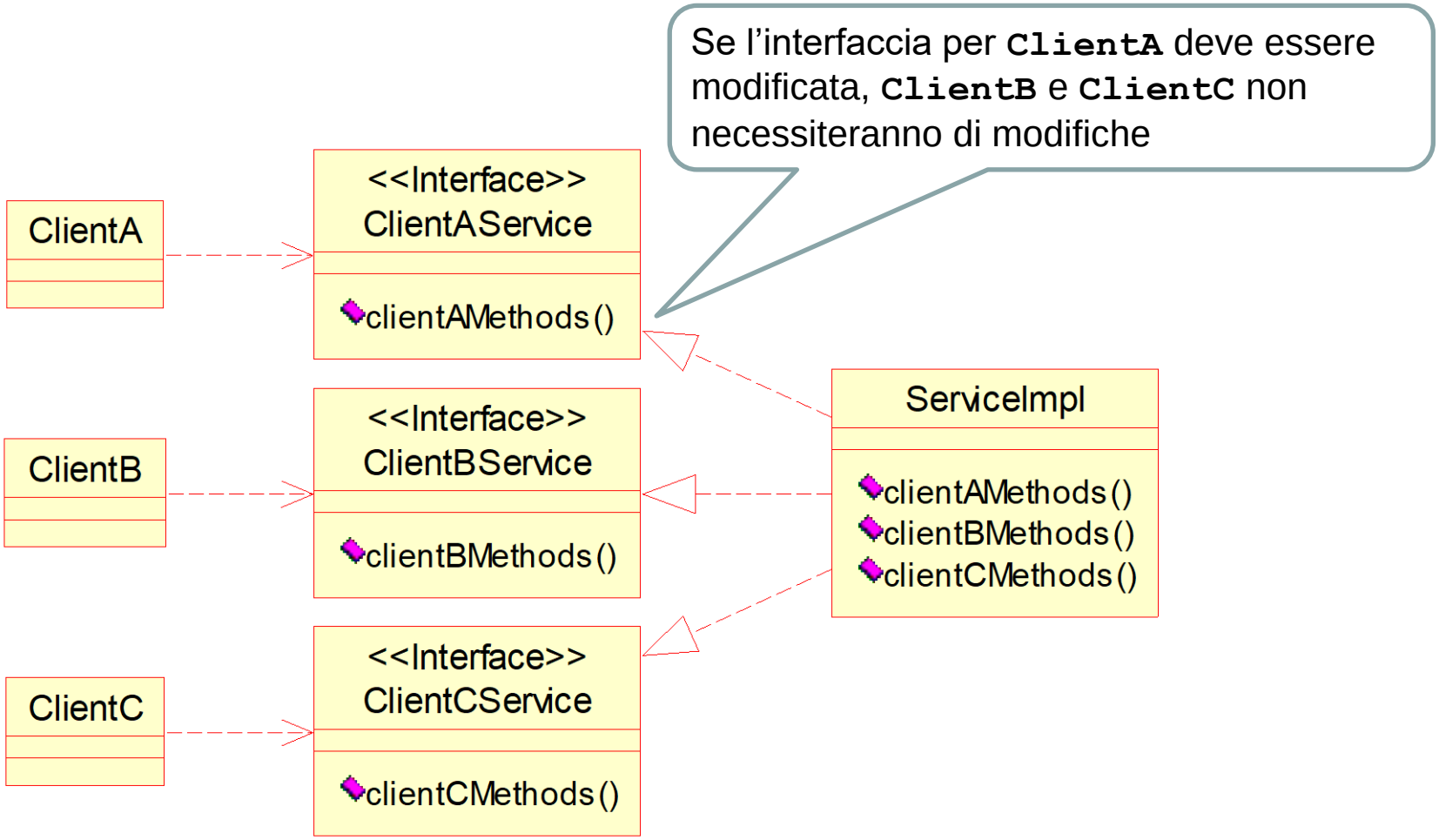
The Interface Segregation Principle

- I clienti non dovrebbero dipendere da servizi che non utilizzano
- Le fat interface creano una forma indiretta di accoppiamento (inutile) fra i clienti – se un cliente richiede l'aggiunta di una nuova funzionalità all'interfaccia, ogni altro cliente è costretto a cambiare anche se non è interessato alla nuova funzionalità
- Questo crea un'inutile sforzo di manutenzione e può rendere difficile trovare eventuali errori

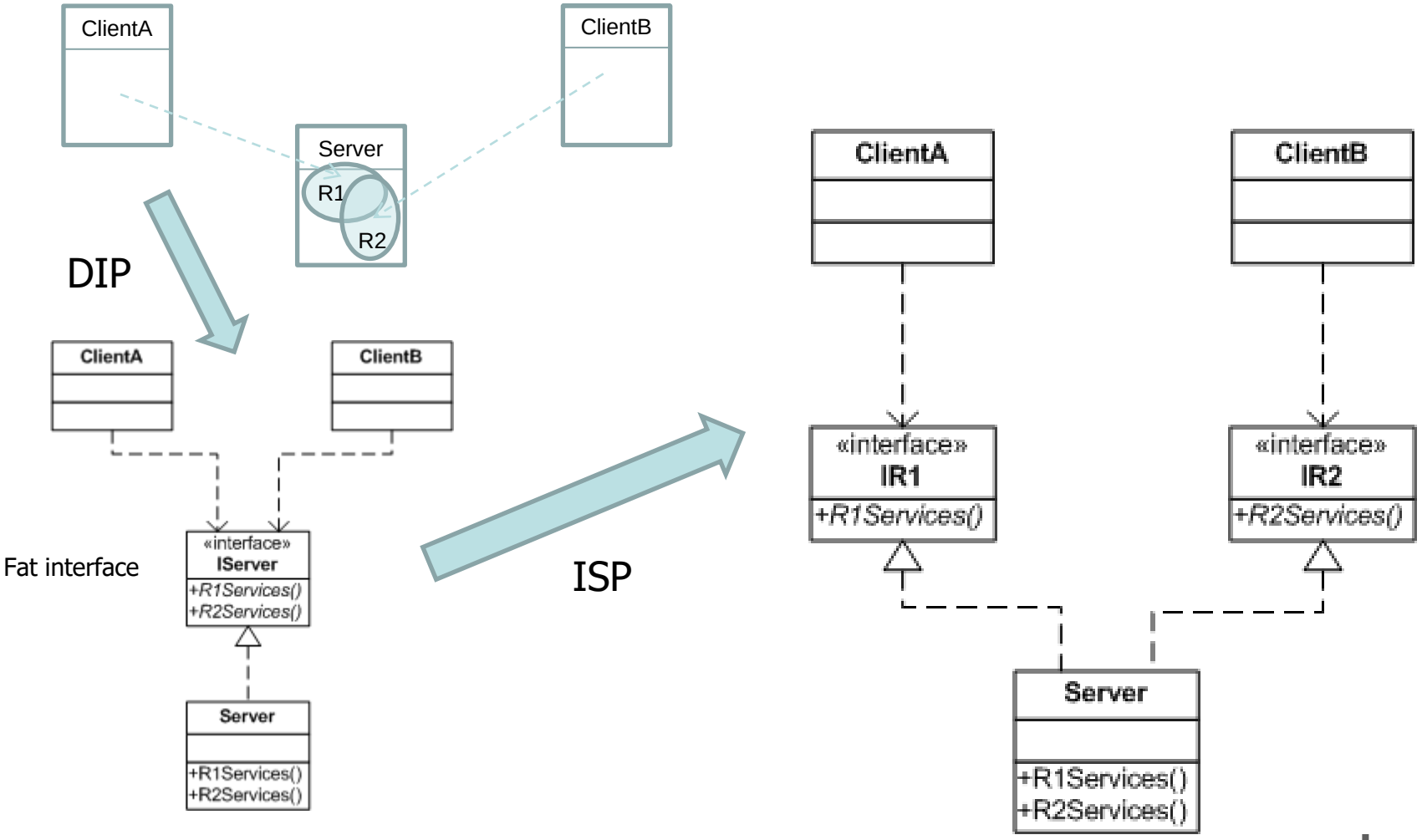
- Se i servizi di una classe possono essere suddivisi in gruppi e ogni gruppo viene utilizzato da un diverso insieme di clienti, creare interfacce specifiche per ogni tipo di cliente e implementare tutte le interfacce nella classe

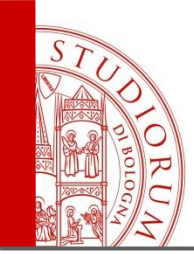
The Interface Segregation Principle

Segregated Interfaces



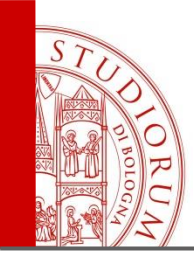
The Interface Segregation Principle





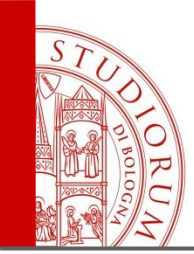
The Open/Closed Principle

- **Il principio più importante** per la **progettazione di entità riutilizzabili**
- ***Software entities (classes, modules, functions, …) should be open for extension, but closed for modification***
- **Open**:
  - **Possono essere estese** aggiungendo nuovo stato o proprietà comportamentali
- **Closed**:
  - Hanno un'interfaccia ben-definita, pubblica e stabile che **non può essere cambiata**



The Open/Closed Principle

- Dobbiamo scrivere i moduli in modo che
  - **possano essere estesi**,
  - **senza** la necessità di **essere modificati**
- In altre parole, vogliamo
  - cambiare quello che fanno i moduli,
  - senza cambiare il codice dei moduli
- Apparentemente si tratta di una **contraddizione**: come può un modulo immutabile esibire un comportamento che non sia fisso nel tempo?
- La risposta risiede **nell'astrazione**: è possibile creare astrazioni che rendono un modulo immutabile, ma rappresentano un gruppo illimitato di comportamenti

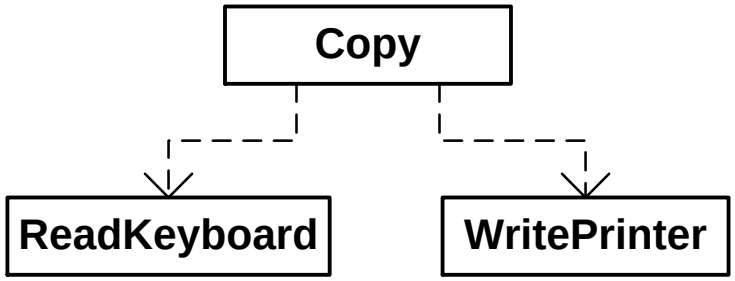


The Open/Closed Principle

- Il segreto sta nell'utilizzo di **interfacce** (o di **classi astratte**)
- A un'interfaccia **immutabile** possono corrispondere innumerevoli classi concrete che realizzano comportamenti diversi
- Un modulo che utilizza astrazioni
  - non dovrà mai essere modificato, dal momento che le astrazioni sono immutabili (**il modulo è chiuso per le modifiche**)
  - potrà cambiare comportamento, se si utilizzano nuove classi che implementano le astrazioni (**il modulo è aperto per le estensioni**)

The Open/Closed Principle
Esempio 1

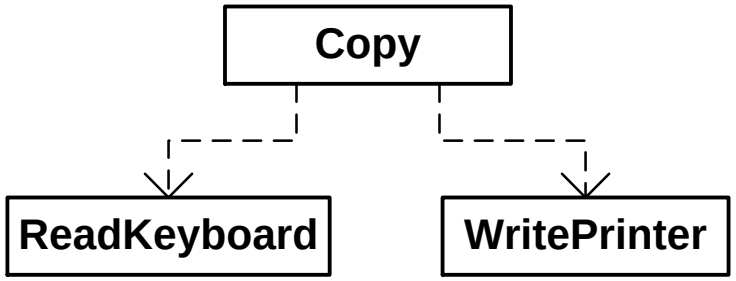
- Consideriamo un semplice programma che si occupa di copiare su una stampante i caratteri digitati su una tastiera
- Assumiamo, inoltre, che la piattaforma di implementazione non possieda un sistema operativo in grado di supportare l'indipendenza dal dispositivo




```
void Copy()
{
int c;
while ((c = ReadKeyboard()) != EOF)
  WritePrinter(c);
}
```

The Open/Closed Principle
Esempio 1

- **I due moduli di basso livello sono riutilizzabili**: possono essere usati in tanti altri programmi per accedere alla tastiera e alla stampante – è la stessa riusabilità offerta dalle librerie di classi
- **Il modulo "Copy" non è riutilizzabile** in un qualsiasi contesto che non includa una tastiera o una stampante
- È un peccato, perché tale modulo contiene "l'intelligenza del sistema" – è il modulo "Copy" che incapsula la funzionalità cui siamo interessati per il riuso




```
void Copy()
{
int c;
while ((c = ReadKeyboard()) != EOF)
  WritePrinter(c);
}
```

The Open/Closed Principle
Esempio 1

- Consideriamo un nuovo programma che copi caratteri da tastiera a un file su disco
- Potremmo modificare il modulo “Copy” per fornirgli questa nuova funzionalità
- Con l’andar del tempo, più e più dispositivi verranno aggiunti a questo programma di copia, e il modulo “Copy” sarà tappezzato di istruzioni if/else, **diventando dipendente da diversi moduli di più basso livello**
  - ▶ alla fine diverrà **rigido** e **fragile**

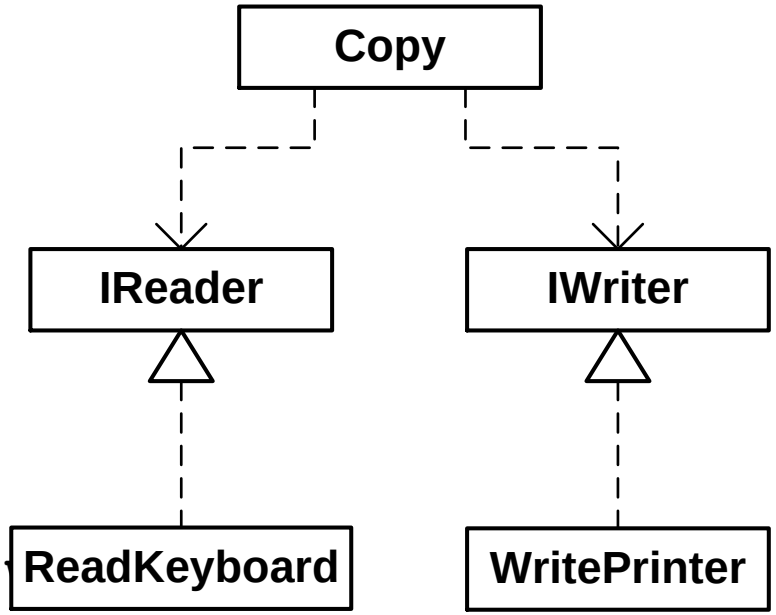
```
enum OutputDevice
{
  Printer,
  Disk
};

void Copy(OutputDevice dev)
{
int c;
while ((c = ReadKeyboard())
   != EOF)
  {
  if (dev == Printer)
    WritePrinter(c);
  else
    WriteDisk(c);
  }
}
```

The Open/Closed Principle
Esempio 1

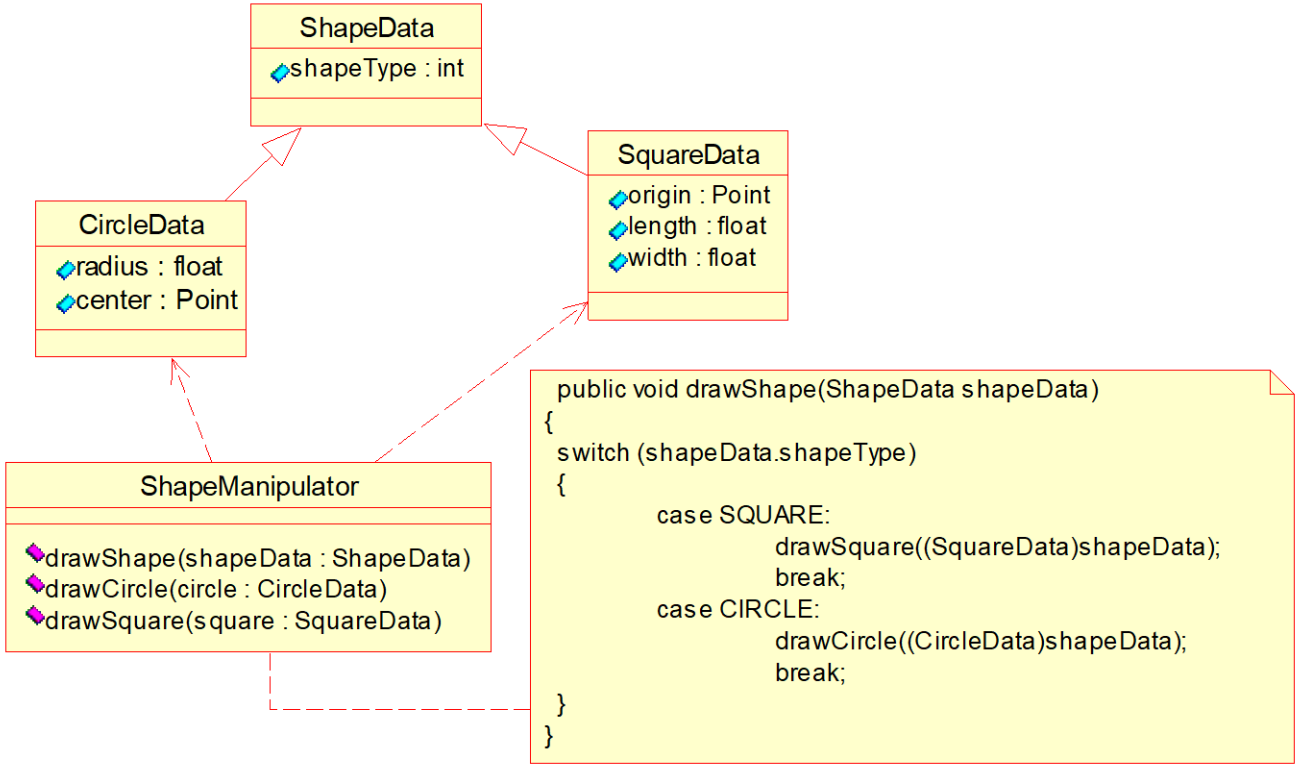
- Un modo per caratterizzare il problema visto in precedenza è di notare che il modulo che contiene la politica di alto livello (Copy) dipende dai moduli di dettaglio e di più basso livello che controlla (WritePrinter e ReadKeyboard)
- Se potessimo trovare un modo di rendere il modulo Copy indipendente dai dettagli che controlla, allora
  - potremmo **riutilizzarlo** liberamente
  - potremmo produrre altri programmi che usano questo modulo per **copiare caratteri da un qualsiasi dispositivo di input a un qualsiasi dispositivo di output**

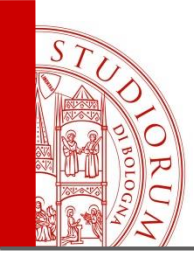
The Open/Closed Principle
Esempio 1

```
interface IReader
{
  int Read();
}

interface IWriter
{
  void Write(char);
}

void Copy(IReader r, IWriter w)
{
int c;
while ((c = r.Read()) != EOF)
  w.Write(c);
}
```



The Open/Closed Principle
Esempio 2

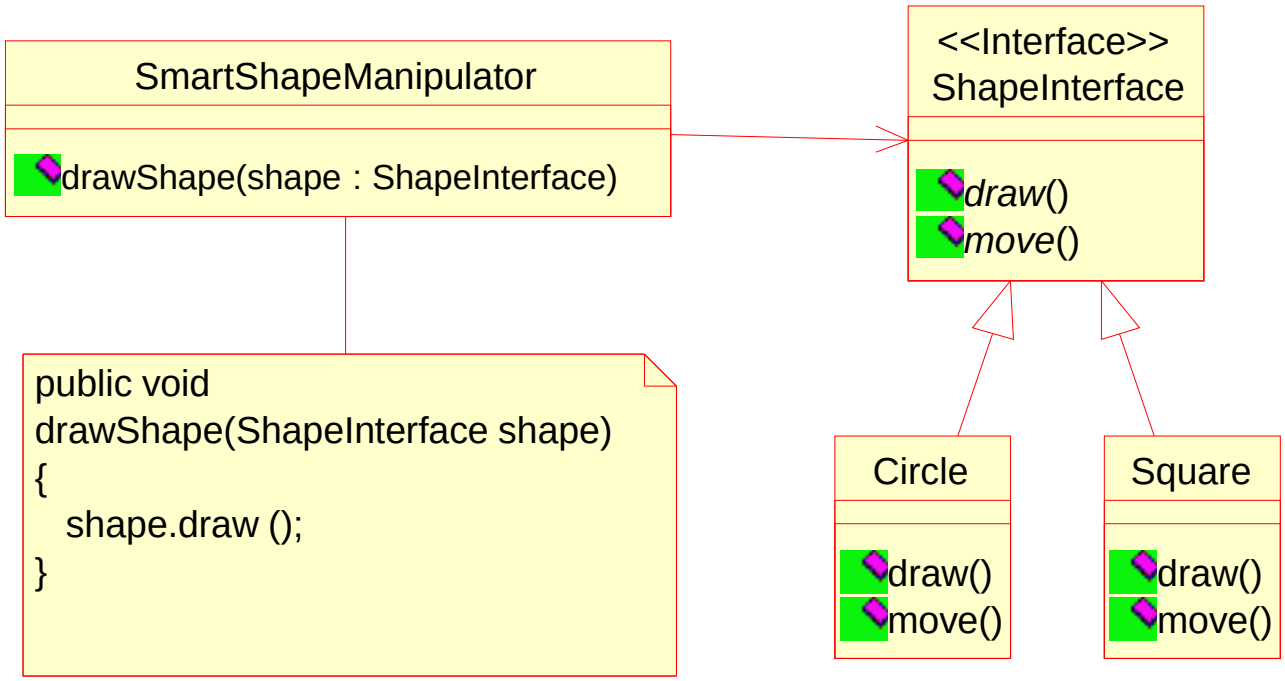




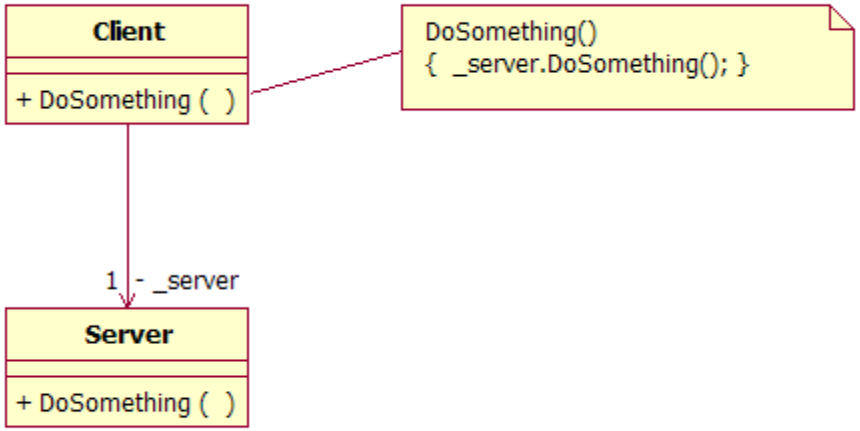
The Open/Closed Principle
Esempio 2

- **Se dovessi creare una nuova** shape, come **`Triangle`**, **dovrei modificare `drawShape`**
- In un'applicazione complessa l'istruzione **`switch/case`** verrebbe ripetuta più e più volte per ogni operazione possa essere effettuata su una shape
- Ancor peggio, ogni modulo che contenesse una tale istruzione **`switch/case`** statement **manterrebbe una dipendenza da qualsiasi shape** possa essere disegnata
  - Quindi, ogni volta una singola shape dovesse essere modificata in un qualsiasi modo, tutti i moduli dovrebbero essere ricompilati ed eventualmente modificati

The Open/Closed Principle
Esempio 2

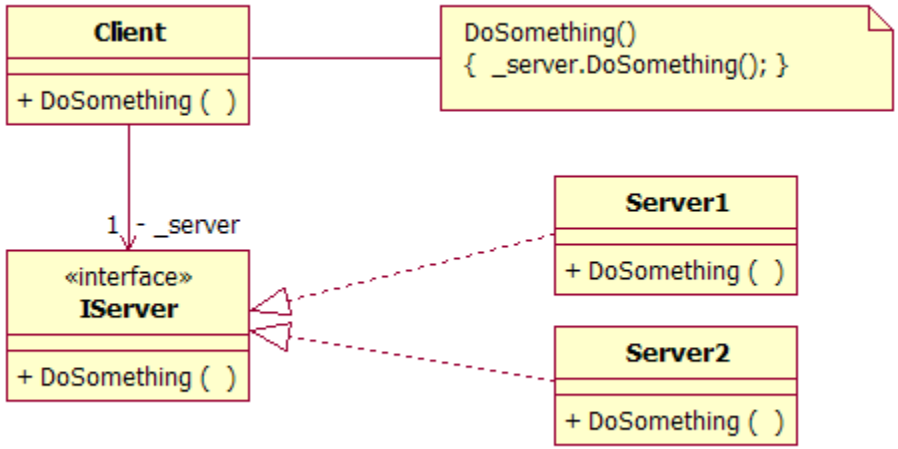


The Open/Closed Principle
Esempio 3



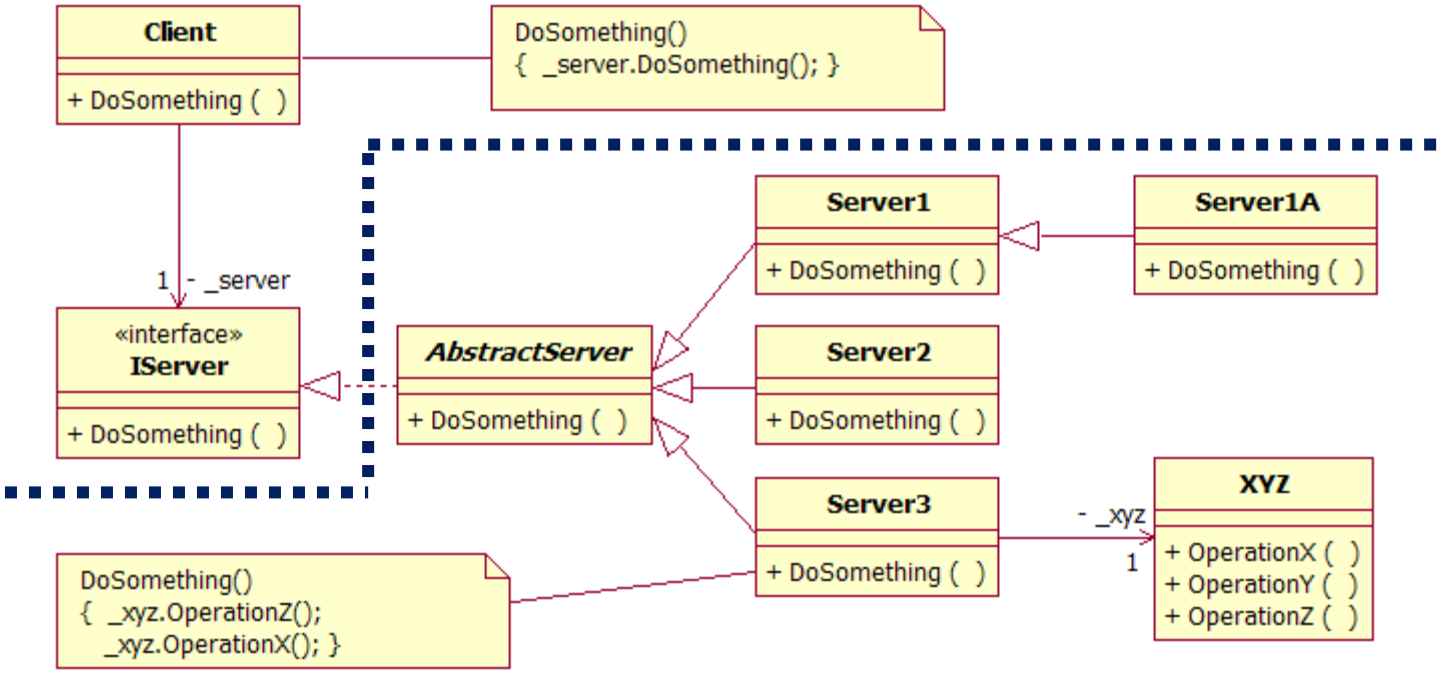
- Supponiamo di dover utilizzare un nuovo tipo di server!

The Open/Closed Principle
Esempio 3

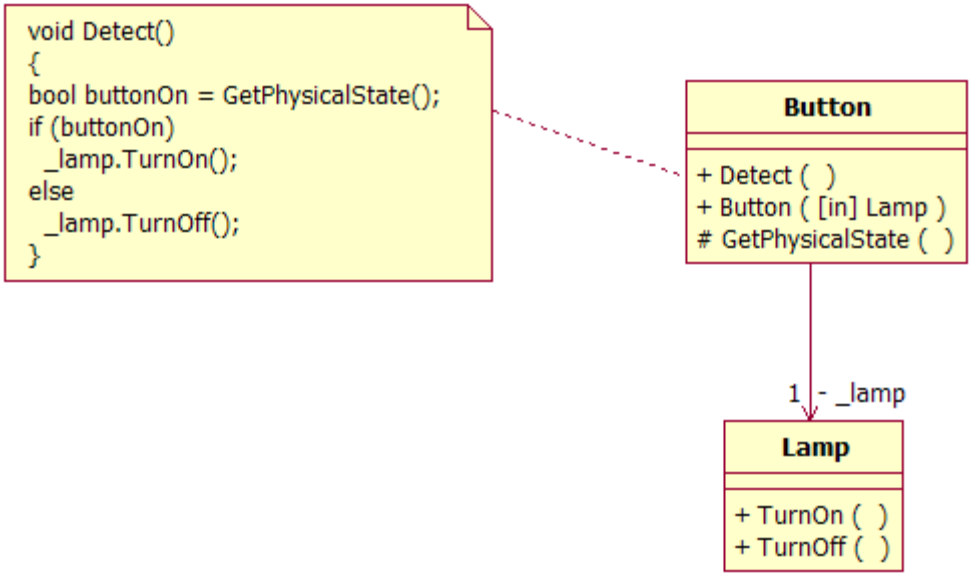


- **`Client`** è chiuso alle modifiche dell'implementazione di **`IServer`**
- **`Client`** è aperto all'estensione tramite nuove implementazioni di **`IServer`**
- Senza **`IServer`**, **`Client`** sarebbe aperto alle modifiche di **`Server`**

The Open/Closed Principle
Esempio 3

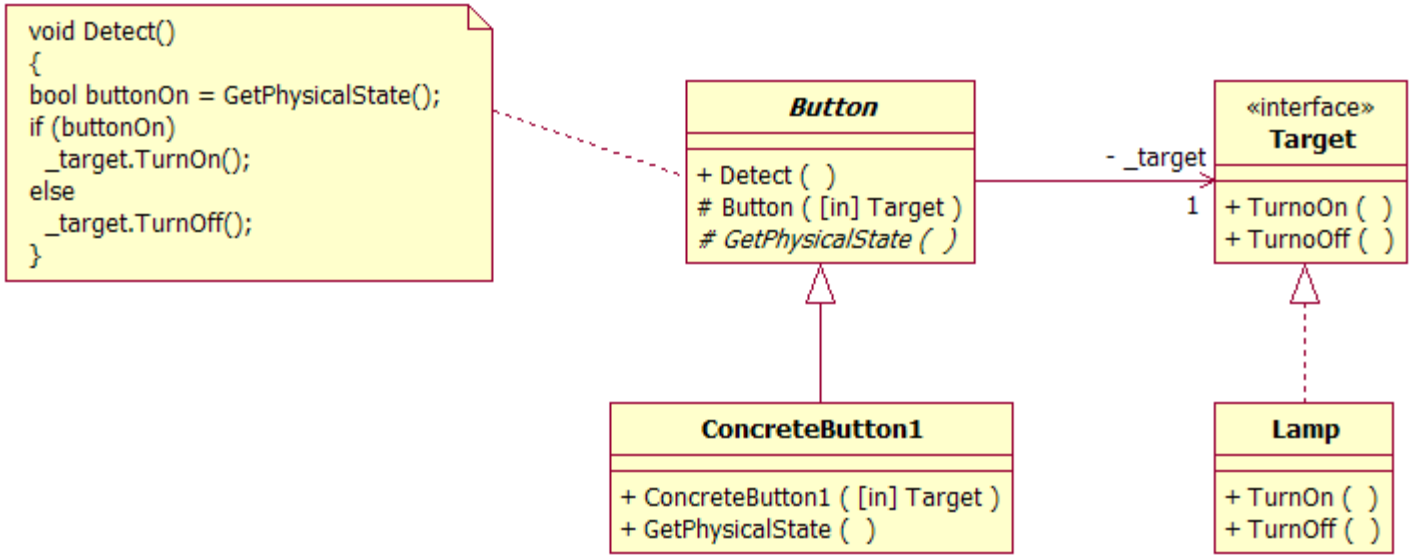


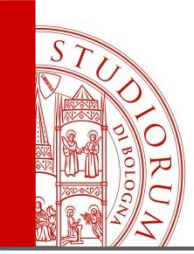
The Open/Closed Principle
Esempio 4



- E se volessimo accendere un motore?

The Open/Closed Principle
Esempio 4





The Open/Closed Principle

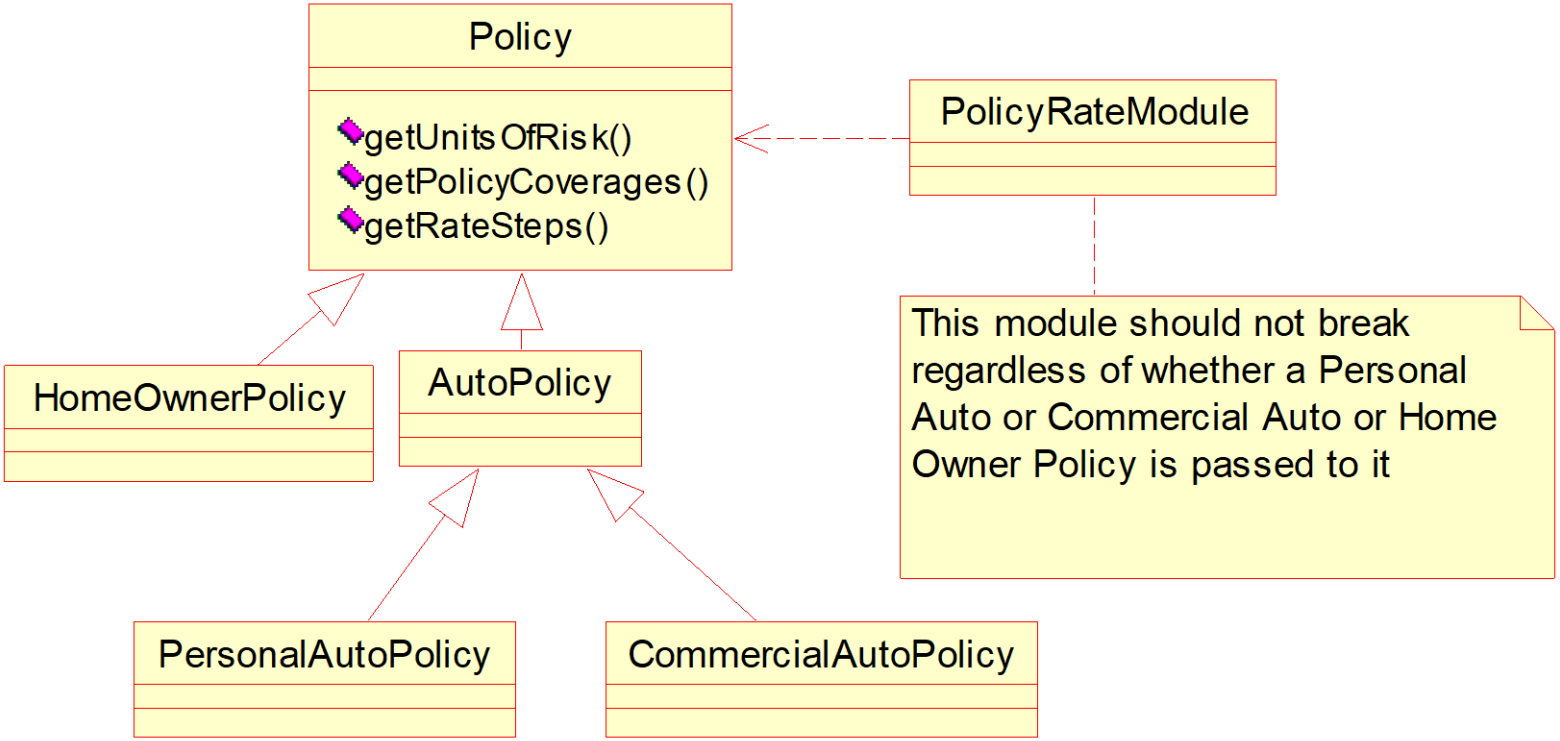
- Se la maggior parte dei moduli di un'applicazione segue OCP, allora
  - è possibile aggiungere nuove funzionalità all'applicazione
    - **aggiungendo nuovo codice**
    - invece che **cambiando codice funzionante**
  - il codice che già funziona non è esposto a rotture

The Liskov Substitution Principle

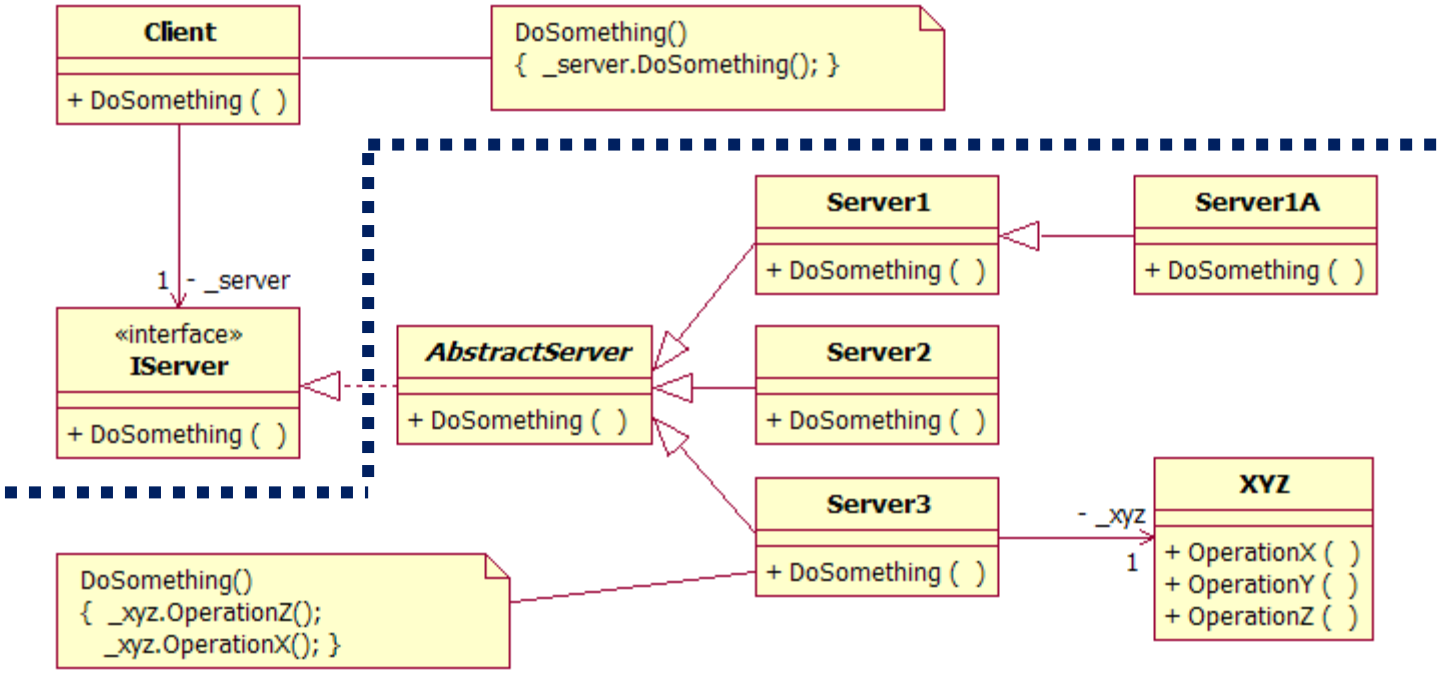
- ***Subclasses should be substitutable for their base classes*** (Barbara Liskov)
- ***All derived classes must honor the contracts of their base classes*** (Design by Contract – Bertrand Meyer)

- Il cliente di una classe base deve continuare a funzionare correttamente se gli viene passato un sottotipo di tale classe base
- In altre parole: un cliente che usa istanze di una classe A deve poter usare istanze di una qualsiasi sottoclasse di A **senza accorgersi della differenza**

The Liskov Substitution Principle Example



The Liskov Substitution Principle Example

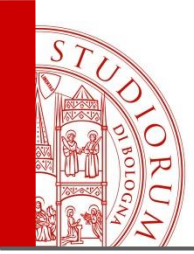


The Liskov Substitution Principle

- OCP si basa sull'uso di classi concrete derivate da un'astrazione (interfaccia o classe astratta)
- LSP costituisce una guida per creare queste classi concrete mediante l'ereditarietà
- La principale causa di violazioni al principio di Liskov è dato dalla **ridefinizione di metodi virtuali** nelle classi derivate:
  è qui che bisogna riporre la massima attenzione
- La chiave per evitare le violazioni al principio di Liskov risiede nel **Design by Contract** (B. Meyer)

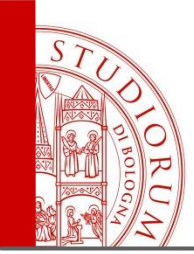
Design by Contract

- Nel Design by Contract ogni metodo ha
  - un insieme di **pre-condizioni** – requisiti minimi che devono essere soddisfatti dal chiamante perché il metodo possa essere eseguito correttamente
  - un insieme di **post-condizioni** – requisiti che devono essere soddisfatti dal metodo nel caso di esecuzione corretta
- Questi due insiemi di condizioni costituiscono **un contratto tra** chi invoca il metodo (**cliente**) **e** il **metodo** stesso (e quindi la classe a cui appartiene)
  - le **pre-condizioni vincolano il chiamante**
  - le **post-condizioni vincolano il metodo**
  - se il chiamante garantisce il verificarsi delle pre-condizioni, il metodo garantisce il verificarsi delle post-condizioni



Design by Contract

- Quando un metodo viene ridefinito in una sottoclasse
  - le **pre-condizioni** devono essere **identiche o meno stringenti**
  - le **post-condizioni** devono essere **identiche o più stringenti**
- Questo perché un cliente che invoca il metodo conosce il contratto definito a livello della classe base, quindi non è in grado:
  - di soddisfare pre-condizioni più stringenti o
  - di accettare post-condizioni meno stringenti
- In caso contrario, il cliente dovrebbe conoscere informazioni sulla classe derivata e questo porterebbe a una violazione del principio di Liskov



Design by Contract

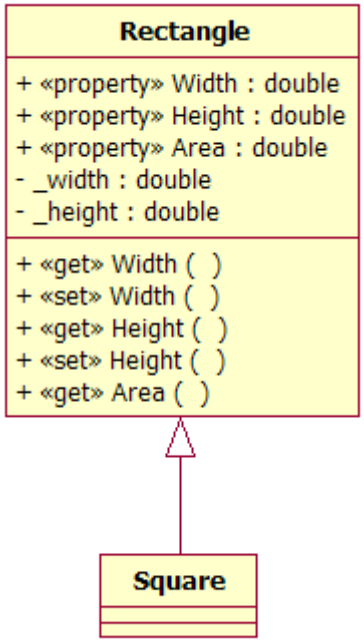
```
public class BaseClass
{
  public virtual int Calculate(int val)
  {
    Precondition(-10000 <= val && val <= 10000);
    int result = val / 100;
    Postcondition(-100 <= result && result <= 100);
    return result;
  }
}

public class SubClass : BaseClass
{
  public override int Calculate(int val)
  {
    Precondition(-20000 <= val && val <= 20000);
    int result = Math.Abs(val) / 200;
    Postcondition(0 <= result && result <= 100);
    return result;
  }
}
```

Il Quadrato è un Rettangolo?

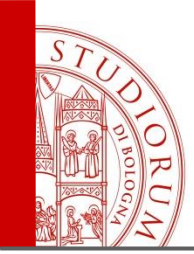
```
public class Rectangle
{
  private double _width;
  private double _height;

  public double Width
  {
    get { return _width; }
    set { _width = value; }
  }

  public double Height
  {
    get { return _height; }
    set { _height = value; }
  }

  public double Area
  {
    get { return Width * Height; }
  }
}
```

- Immaginiamo che questa applicazione funzioni correttamente e sia installata in diversi ambienti
- Come per tutti i software di successo, le necessità degli utenti cambiano e si rendono necessarie nuove funzionalità
- Immaginiamo che, un bel giorno, gli utenti chiedano la possibilità di manipolare quadrati oltre che rettangoli

Il Quadrato è un Rettangolo?



- L'ereditarietà è una relazione IsA
- In altre parole, perché un nuovo tipo di oggetto verifichi la relazione IsA con un tipo di oggetto esistente, la classe del nuovo oggetto deve essere derivata dalla classe dell'oggetto esistente
- Chiaramente, un quadrato è un rettangolo per tutti gli utilizzi e intenti normali
- Poiché vale la relazione IsA, è logico medellare **`Square`** come sottoclasse di **`Rectangle`**



Il Quadrato è un Rettangolo?

- Questo utilizzo della relazione IsA è considerato da molti come una delle tecniche più importanti dell'Analisi Object Oriented
- Un quadrato è un tipo particolare di rettangolo, quindi la classe **`Square`** *deve* venire derivata dalla classe **`Rectangle`**
- Questo modo di pensare, però, può portare a problemi sottili, ma significativi
- In genere, tali problemi non vengono scoperti se non nella fase di implementazione dell'applicazione

Il Quadrato è un Rettangolo?

```
public class Square : Rectangle
{
  public new double Width
  {
    get { return base.Width; }
    set
    {
          base.Width = value;
          base.Height = value;
    }
  }

  public new double Height
  {
    get { return base.Height; }
    set
    {
          base.Height = value;
          base.Width = value;
    }
  }
}
```

- È necessario ridefinire le proprietà **`Width`** e **`Height`**…
- Notevoli differenze tra Java e C++/C#
  - In Java tutti i metodi sono virtuali
    - a parte i metodi `final`
  - In C++ / C# è possibile definire
    - sia metodi virtuali,
    - sia **metodi non virtuali** (non polimorfici)

Il Quadrato è un Rettangolo?

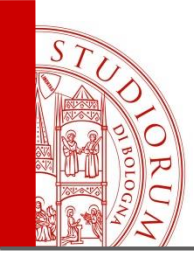
```
…
Square s = new Square();
s.Width = 5;  //  5 x 5
…
… Method1(s);  //  ?
…

void Method1(Rectangle r)
{
  r.Width = 10;
}
```

- Se invochiamo **`Method1`** con un riferimento a un oggetto **`Square`**, l'oggetto **`Square`** non funzionerà correttamente in quanto l'altezza non verrà modificata (!)
- Questa è una chiara violazione di LSP
- **`Method1`** non funziona per sottotipi dei suoi parametri
- **Il motivo di questo malfunzionamento è che `Width` e `Height` non sono state dichiarate virtuali in `Rectangle`**

Il Quadrato è un Rettangolo?

```
public class Rectangle
{
  …
  public virtual double Width
  { … }
  public virtual double Height
  { … }
  …
}

public class Square : Rectangle
{
  public override double Width
  { … }
  public override double Height
  { … }
}
```

- Possiamo risolvere facilmente
- In ogni modo, **quando la creazione di una classe derivata ci obbliga a modificare la classe base, spesso significa che il design è difettoso**
- Infatti, viola l'OCP
- Potremmo rispondere argomentando che il vero difetto di progettazione è stato dimenticare di rendere virtuali **`Width`** e **`Height`**, e solo ora lo stiamo risolvendo
- Tuttavia, questo è difficile da giustificare poiché impostare l'altezza e la larghezza di un rettangolo sono operazioni estremamente primitive – con quale ragionamento le avremmo dovute rendere virtuali se non prevedendo l'esistenza del quadrato?



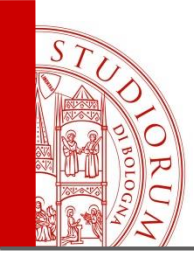
Il Quadrato è un Rettangolo?

- A questo punto abbiamo due classi, **`Square`** e **`Rectangle`**, che apparentemente funzionano correttamente
- Indipendentemente da ciò che facciamo con un oggetto **`Square`**, questo rimarrà coerente con un quadrato matematico
- E indipendentemente da ciò che facciamo con un oggetto **`Rectangle`**, questo rimarrà un rettangolo matematico
- Inoltre, possiamo passare uno **`Square`** a una funzione che accetta un riferimento a un **`Rectangle`** e lo **`Square`** agirà comunque come un quadrato e rimarrà consistente
- Pertanto, potremmo concludere che il modello ora è consistente e corretto in sé
- Tuttavia, **un modello consistente in sé non è necessariamente consistente con tutti i suoi utenti**!

Il Quadrato è un Rettangolo?

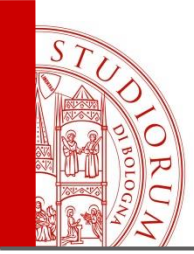
```
public void Scale(Rectangle rectangle)
{
  rectangle.Width = rectangle.Width * ScalingFactor;
  rectangle.Height = rectangle.Height * ScalingFactor;
}
```

- **`Scale`** invoca membri di ciò che crede essere un **`Rectangle`**
- Sostituendovi uno **`Square`** otterremo che il quadrato verrà ridimensionato due volte!
- E allora qui sta **il vero problema**: il programmatore che ha scritto **`Scale`** era giustificato nel presumere che la modifica della larghezza di un **`Rectangle`** lasci invariata la sua altezza?



Il Quadrato è un Rettangolo?

- Chiaramente, il programmatore di **`Scale`** ha fatto questa ipotesi assai ragionevole
- Passare uno **`Square`** a funzioni i cui programmatori hanno fatto questa ipotesi provocherà problemi
- Pertanto, esistono funzioni che accettano riferimenti a oggetti **`Rectangle`** , ma non possono operare correttamente su oggetti **`Square`**
- Queste funzioni espongono una violazione di LSP
- L'aggiunta del sottotipo **`Square`** di **`Rectangle`** ha guastato queste funzioni ▶ l'OCP è stato violato

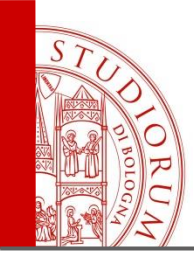


Il Quadrato è un Rettangolo?

- Cosa non va nel modello di **`Square`** e **`Rectangle`**?
  - Dopo tutto, un quadrato non è un rettangolo?
  - La relazione IsA non vale?
- No! Un quadrato sarà anche un rettangolo, ma un oggetto **`Square`** non è sicuramente un oggetto **`Rectangle`**
- Perché? Perché **il comportamento di un oggetto `Square` non è consistente con il comportamento di un oggetto `Rectangle`**
- Dal punto di vista comportamentale, uno **`Square`** non è un **`Rectangle`**! E il software si basa proprio sul comportamento

Il Quadrato è un Rettangolo?

- LSP chiarisce che **in OOD la relazione IsA riguarda il comportamento**
  - Non comportamento privato intrinseco, ma **comportamento pubblico estrinseco**
  - Comportamento da cui dipendono i clienti



Il Quadrato è un Rettangolo?

- Ad esempio, l'autore di **`Scale`** dipendeva dal fatto che i rettangoli si comportano in modo tale che **le loro altezza e larghezza possano variare indipendentemente l'una dall'altra**
- Tale indipendenza delle due variabili è un **comportamento pubblico estrinseco** da cui probabilmente dipenderanno altri programmatori
- Affinché LSP possa valere, e con esso OCP, **tutti i sottotipi devono essere conformi al comportamento che i clienti si aspettano dalle classi base che utilizzano**

Il Quadrato è un Rettangolo?

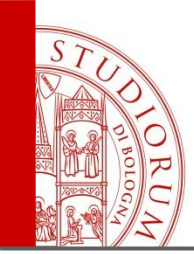
- La regola per le pre-condizioni e le post-condizioni per i sottotipi è:

  "quando si ridefinisce una routine, si può sostituire **la sua pre-condizione solo con una più debole** e **la sua post-condizione solo con una più forte**"

- In altre parole, quando si utilizza un oggetto attraverso l'interfaccia della sua classe base, **l'utente conosce solo le pre-condizioni e le post-condizioni della classe base**

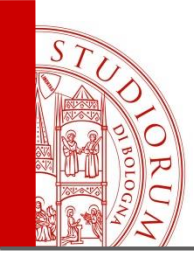
Il Quadrato è un Rettangolo?

- Pertanto, le classi derivate non devono aspettarsi che tali utenti obbediscano a pre-condizioni più forti di quelle richieste dalla classe base
  ▶ devono accettare tutto ciò che la classe base può accettare
- Inoltre, le classi derivate devono essere conformi a tutte le post-condizioni della classe base
  ▶ i loro comportamenti e output non devono violare nessuno dei vincoli stabiliti per la classe base



Il Quadrato è un Rettangolo?

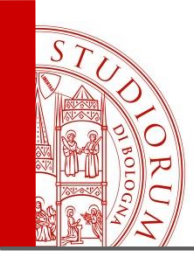
- Il contratto di **`Rectangle`**
  - Altezza e larghezza sono indipendenti, si può modificarne una mantenendo costante l'altra
- **`Square`** viola il contratto della classe base



Il Quadrato è un Rettangolo?

- Guardando al codice di test della classe **`Rectangle`** possiamo avere qualche idea del contratto di **`Rectangle`**:

```
[TestFixture]
public class RectangleFixture
{
  [Test]
  public void SetHeightAndWidth()
  {
    Rectangle rectangle = new Rectangle();
    int expectedWidth = 3, expectedHeight = 7;
    rectangle.Width = expectedWidth;
    rectangle.Height = expectedHeight;
    Assertion.AssertEquals(expectedWidth, rectangle.Width);
    Assertion.AssertEquals(expectedHeight, rectangle.Height);
  }
}
```



Il Quadrato è un Rettangolo?

```
[TestFixture] public class RectangleFixture
{
  [Test] public void SetHeightAndWidth()
  {
    Rectangle rectangle = GetShape();
    ...
  }
  protected virtual Rectangle GetShape()
  { return new Rectangle(); }
}

[TestFixture] public class SquareFixture : RectangleFixture
{
  protected override Rectangle GetShape()
  { return new Square(); }
}
```

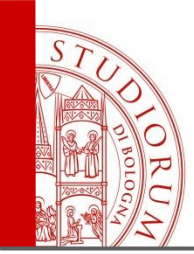
Principi di Architettura dei Package

- **Reuse/Release Equivalency Principle** (REP)
- **Common Closure Principle** (CCP)
- **Common Reuse Principle** (CRP)

Release/Reuse Equivalency Principle

- ***The granule of reuse is the granule of release***

- Un elemento riutilizzabile, sia esso un componente, una classe o un insieme di classi, non può essere riutilizzato a meno che non sia gestito da un sistema di rilascio di qualche tipo
- I clienti dovrebbero rifiutare di riutilizzare un elemento a meno che l'autore non prometta di tenere traccia dei numeri di versione e di mantenere le vecchie versioni per qualche tempo
  - ▶ un criterio per raggruppare le classi in package è il riutilizzo
- Poiché i pacchetti sono l'unità di rilascio in Java, sono anche l'unità di riutilizzo
- Pertanto, gli architetti farebbero bene a raggruppare in package le classi riutilizzabili assieme



Common Closure Principle

- ***Classes that change together, belong together***

- Il lavoro per gestire, testare e rilasciare un pacchetto in un sistema di grandi dimensioni non è banale
- Più pacchetti cambiano in un dato rilascio, maggiore è il lavoro per ricostruire, testare e distribuire il rilascio
  ▶ vorremmo **ridurre al minimo il numero di pacchetti che vengono modificati in un dato ciclo di rilascio del prodotto**
- Per raggiungere questo obiettivo, raggruppiamo assieme classi che pensiamo cambieranno insieme

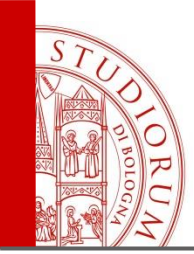
Common Reuse Principle

- ***Classes that aren't reused together should not be grouped together***

- Una dipendenza da un package è una dipendenza da tutto ciò che è contenuto nel package
- Quando un package cambia e il suo numero di rilascio viene aggiornato, tutti i client di quel package devono verificare di funzionare con il nuovo package – anche se nulla di ciò che hanno usato all'interno del package è effettivamente cambiato
- Pertanto, le classi che non vengono riutilizzate insieme non dovrebbero essere raggruppate insieme

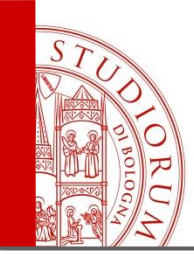
Principi di Architettura dei Package
Discussione

- Questi tre principi non possono essere soddisfatti contemporaneamente
- REP e CRP semplificano la vita ai riutilizzatori, mentre CCP semplifica la vita ai manutentori
- CCP cerca di rendere i package più grandi possibile (dopotutto, se tutte le classi stanno in un solo package, allora solo quel package cambierà)
- CRP, tuttavia, cerca di creare package molto piccoli
- All'inizio di un progetto, gli architetti possono impostare la struttura dei package in modo tale che CCP domini per facilità di sviluppo e manutenzione
- Successivamente, quando l'architettura si stabilizza, gli architetti possono ri-fattorizzare la struttura dei package per massimizzare REP e CRP per i riutilizzatori esterni



Relazioni tra i Package

- **Acyclic Dependencies Principle** (ADP)
- **Stable Dependencies Principle** (SDP)
- **Stable Abstractions Principle** (SAP)

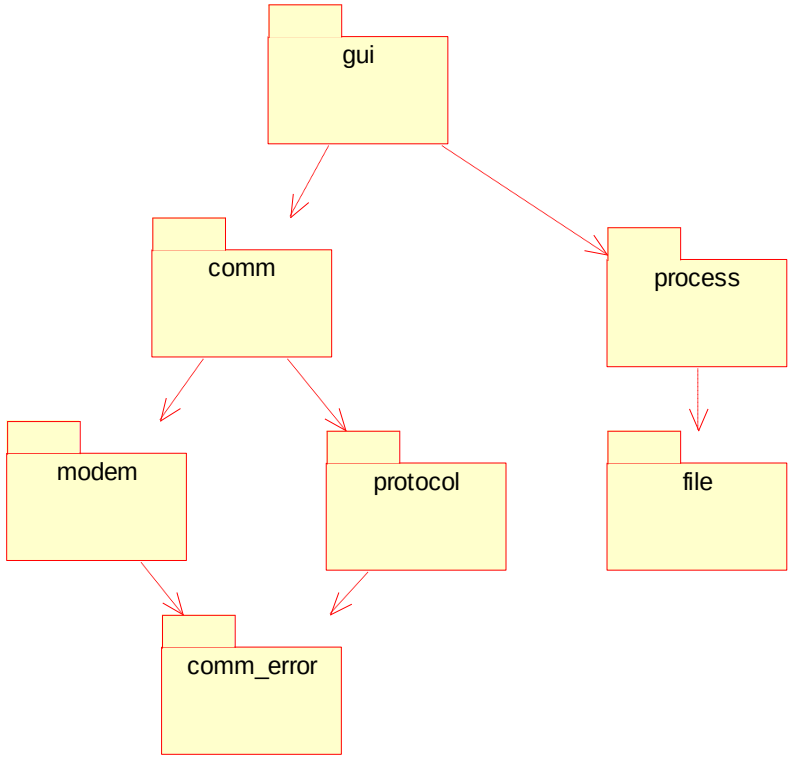


Acyclic Dependencies Principle

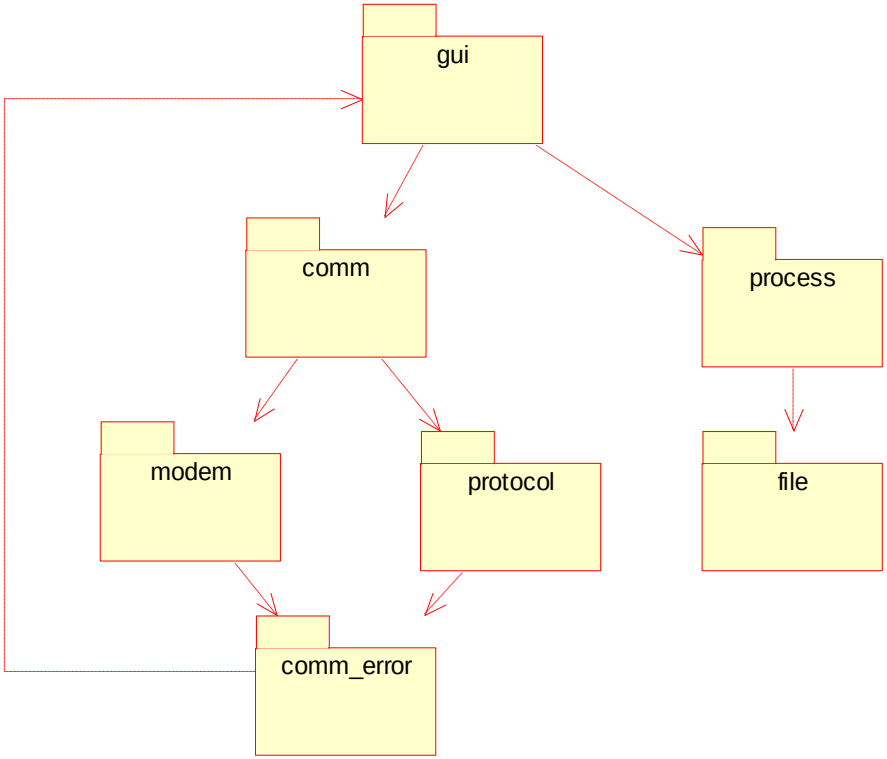
- ***The dependencies between packages must not form cycles***

- Una volta apportate le modifiche a un package, gli sviluppatori possono rilasciare i package al resto del progetto
  - Prima di poter eseguire questo rilascio, tuttavia, devono verificare che il package funzioni
  - Per farlo, devono compilarlo e collegarlo a tutti i package da cui dipende
- Una singola dipendenza ciclica che sfugge al controllo può rendere l'elenco delle dipendenze molto lungo
- Quindi, qualcuno deve osservare la struttura delle dipendenze dei package con regolarità e interrompere i cicli ovunque compaiano

Acyclic Dependencies Principle
Esempio: Grafo dei Package Aciclico



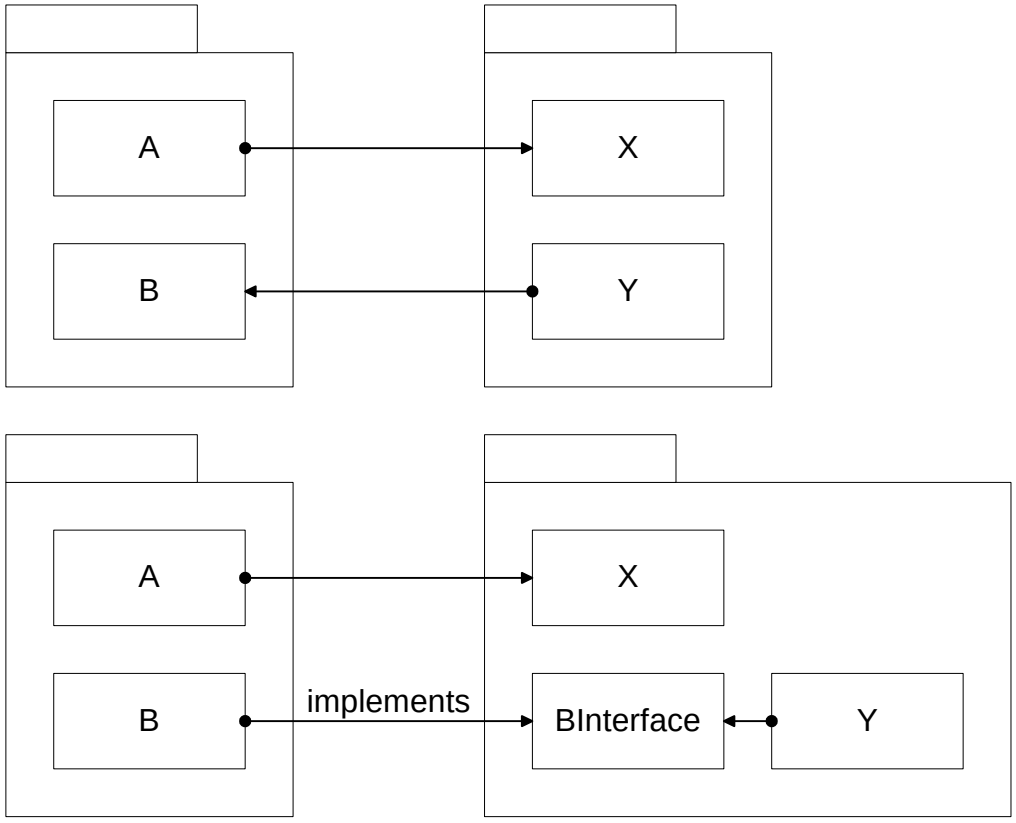
Acyclic Dependencies Principle
Esempio: Grafo dei Package Ciclico



Acyclic Dependencies Principle
Discussione

- Nello scenario aciclico per rilasciare il package protocol, gli ingegneri dovrebbero compilarlo con l'ultima versione del package comm_error ed eseguire i loro test
- Nello scenario ciclico per rilasciare il package protocol, gli ingegneri dovrebbero compilarlo con l'ultima versione di comm_error, gui, comm, process, modem, file ed eseguire i loro test
- Come rompere un ciclo:
  - Inframezzare un nuovo package
  - Aggiungere una nuova interfaccia

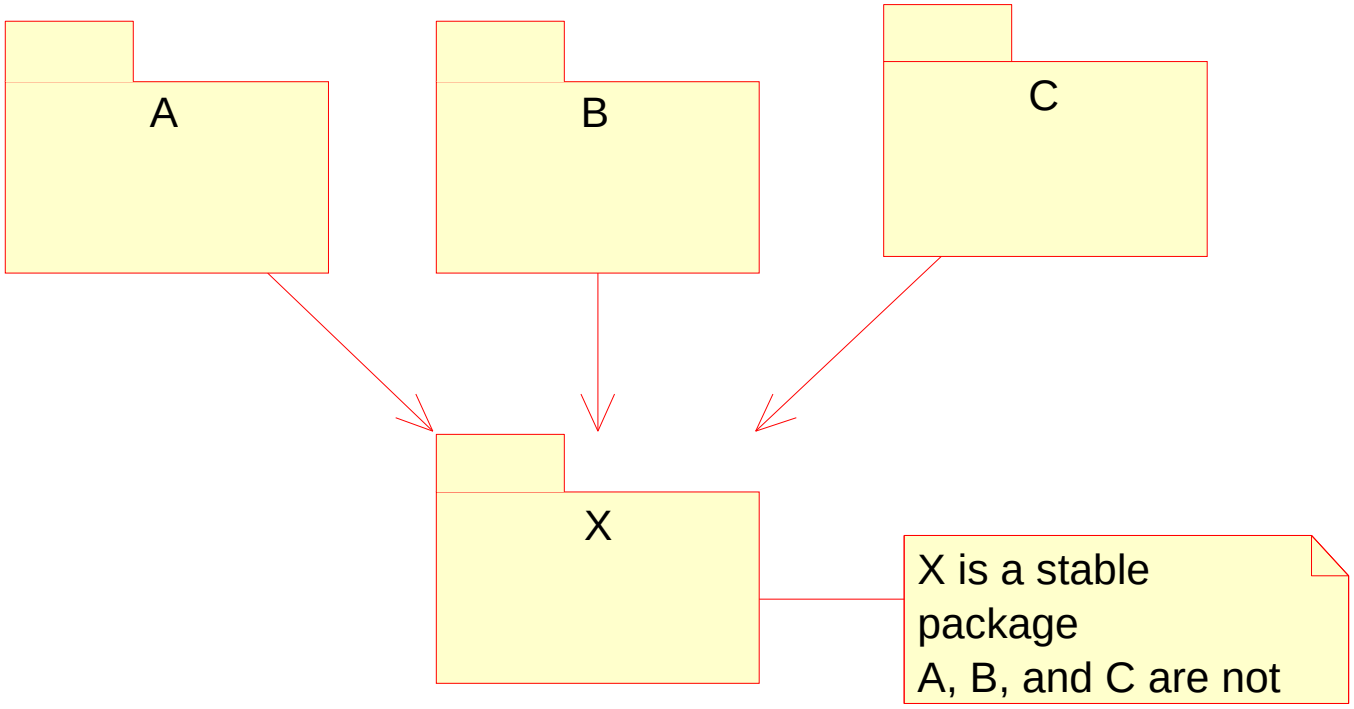
Acyclic Dependencies Principle
Rompere il Ciclo Introducendo un'Interfaccia



Stable Dependencies Principle

- ***The dependencies between packages in a design should be in the direction of the stability of the packages.***
  A package should only depend upon packages that are more stable than it is.
- I design non possono essere completamente statici
  - Una certa volatilità è necessaria se il progetto deve essere mantenuto
- Raggiungiamo questo obiettivo conformandoci al CCP
- Alcuni package sono progettati per essere volatili, ci aspettiamo che cambino
- Un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare qualsiasi modifica con tutti i pacchetti dipendenti

Stable Dependencies Principle
Esempio

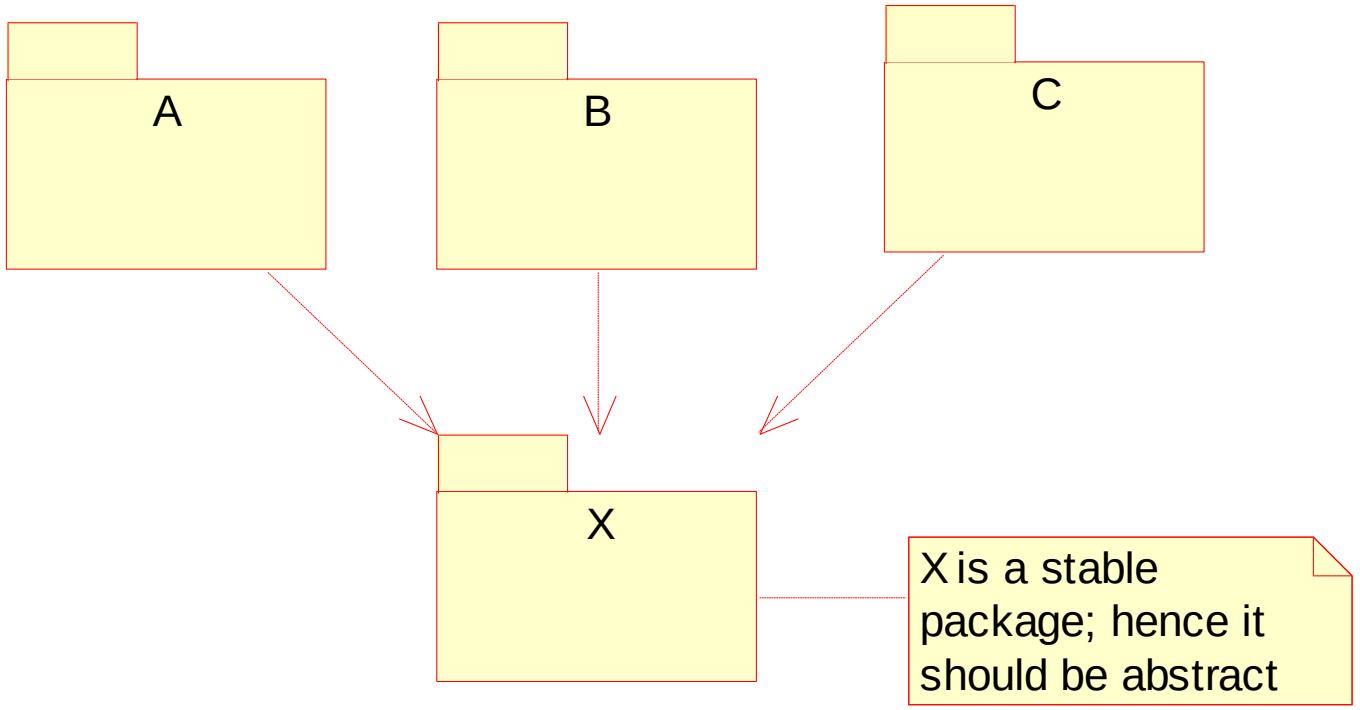


Stable Abstractions Principle

- ***Stable packages should be abstract packages***

- La stabilità è relativa alla quantità di lavoro richiesta per apportare una modifica
- Un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare le modifiche con tutti i pacchetti dipendenti

Stable Abstractions Principle
Esempio



Stable Dependencies/Abstractions Principles
Discussione

- I package in alto sono instabili e flessibili
- I package in basso sono molto difficili da modificare
- I package altamente stabili nella parte inferiore del grafo delle dipendenze possono essere molto difficili da modificare, ma secondo l'OCP non devono essere difficili da estendere
- Se anche i pacchetti stabili in fondo sono molto astratti, possono essere facilmente estesi
- È possibile creare la nostra applicazione a partire da package instabili facili da modificare e package stabili facili da estendere